QuantiPhy Documentation

Release 2.20

Ken Kundert

Apr 27, 2024

CONTENTS

Version: 2.20
Released: 2024-04-27
Please post all bugs and suggestions at Github (or contact me directly at quantiphy@nurdletech.com).

CHAPTER

ONE

WHAT?

QuantiPhy is a Python library that offers support for physical quantities. A quantity is the pairing of a number and a unit of measure that indicates the amount of some measurable thing. *QuantiPhy* provides quantity objects that keep the units with the number, making it easy to share them as single object. They subclass float and so can be used anywhere a real number is appropriate.

CHAPTER

TWO

WHY?

QuantiPhy naturally supports SI scale factors, which are widely used in science and engineering. SI scale factors make it possible to cleanly represent both very large and very small quantities in a form that is both easy to read and write. While generally better for humans, no general programming language provides direct support for reading or writing quantities with SI scale factors, making it difficult to write numerical software that communicates effectively with people. *QuantiPhy* addresses this deficiency, making it natural and simple to both input and output physical quantities.

CHAPTER

THREE

FEATURES

- Flexibly reads amounts with units and SI scale factors.
- Quantities subclass the *float* class and so can be used as conventional numbers.
- Generally includes the units when printing or converting to strings and by default employs SI scale factors.
- Flexible unit conversion and scaling is supported to make it easy to convert to or from any required form.
- Supports the binary scale factors (*Ki*, *Mi*, etc.) along with the normal SI scale factors (*k*, *M*, etc.).
- When a quantity is created from a string, the actual digits specified can be used in any output, eliminating any loss of precision.

CHAPTER

FOUR

ALTERNATIVES

There are a considerable number of Python packages dedicated to units and quantities (alternatives). However, as a rule, they focus on the units rather than the scale factors. In particular, they build a system of units that you are expected to use throughout your calculations. These packages demand a high level of commitment from their users and in turn provide unit consistency and built-in unit conversions.

In contrast, *QuantiPhy* treats units basically as documentation. They are simply strings that are attached to quantities largely so they can be presented to the user when the values are printed. As such, *QuantiPhy* is a light-weight package that demands little from the user. It is used when inputting and outputting values, and then only when it provides value. As a result, it provides a simplicity in use that cannot be matched by the other packages.

In addition, these alternative packages generally build their unit systems upon the SI base units, which tends to restrict usage to physical quantities with static conversion factors. They are less suited to non-physical quantities or conversion factors that change dynamically, such as with currencies. *QuantiPhy* gracefully handles all of these cases.

CHAPTER

FIVE

QUICK START

Install with:

```
pip3 install quantiphy
```

Requires Python 3.6 or newer. If you using an earlier version of Python, install version 2.10 of *QuantiPhy*.

You use *Quantity* to convert numbers and units in various forms to quantities:

```
>>> from quantiphy import Quantity

>>> Tclk = Quantity(10e-9, 's')
>>> print(Tclk)
10 ns

>>> Fhy = Quantity('1420.405751786 MHz')
>>> print(Fhy)
1.4204 GHz

>>> Rsense = Quantity('1e-4')
>>> print(Rsense)
100 u

>>> cost = Quantity('$11_200_000')
>>> print(cost)
$11.2M

>>> Tboil = Quantity('212 °F', scale='°C')
>>> print(Tboil)
100 °C
```

Once you have a quantity, there are a variety of ways of accessing aspects of the quantity:

```
>>> Tclk.real
1e-08

>>> float(Fhy)
1420405751.786

>>> 2*cost
22400000.0
```

(continues on next page)

(continued from previous page)

```
>>> Rsense.units
''

>>> str(Tboil)
'100 °C'
```

You can use the *render* method to flexibly convert the quantity to a string:

```
>>> Tclk.render()
'10 ns'

>>> Tclk.render(show_units=False)
'10n'

>>> Tclk.render(form='eng', show_units=False)
'10e-9'

>>> Fhy.render(prec=8)
'1.42040575 GHz'

>>> Tboil.render(scale='°F')
'212 °F'
```

The *fixed* method is a variant that specializes in rendering numbers without scale factors or exponents:

```
>>> cost.fixed(prec=2, show_commas=True, strip_zeros=False)
'$11,200,000.00'
```

You can use the string format method or the new format strings to flexibly incorporate quantity values into strings:

```
>>> f'{Fhy}'
'1.4204 GHz'

>>> f'{Fhy:.6}'
'1.420406 GHz'

>>> f'{Fhy:<15.6}'
'1.420406 GHz   '

>>> f'{Fhy:>15.6}'
'   1.420406 GHz'

>>> f'{cost:#,.2P}'
'$11,200,000.00'

>>> f'Boiling point of water: {Tboil:s}'
'Boiling point of water: 100 °C'

>>> f'Boiling point of water: {Tboil:s°F}'
'Boiling point of water: 212 °F'
```

CHAPTER

SIX

ISSUES

Please ask questions or report problems on Github.

CHAPTER

SEVEN

DOCUMENTATION

7.1 Users' Guide

7.1.1 Overview

QuantiPhy adds support for quantities to Python. Quantities are little more than a number combined with its units. They are used to represent physical quantities. Your height and weight are both quantities, having both a value and units, and both are important. For example, if I told you that Mariam's weight was 8, you might assume pounds as the unit of measure if you lived in the US and think Mariam was an infant, or you might assume stones as the units if you live in the UK and assume that she was an adult, or you might assume kilograms if you lived anywhere else and assume she was a small child. The units are very important, and in general it is always best to keep the unit of measure with the number and present the complete value when working with quantities. To do otherwise invites confusion. Just ask NASA. Readers often stumble on numbers without units as they mentally try to determine the units from context. Quantity values should be treated in a manner similar to money, which is also a quantity. Monetary amounts are almost always given with their units (a currency symbol).

Having a single object represent a quantity in a programming language is useful because it binds the units to the number making it more likely that the units will be presented with the number. In addition, quantities from *QuantiPhy* provide another important benefit. They naturally support the SI scale factors, which for those that are familiar with them are much easier to read and write than the alternatives. The most common SI scale factors are:

T (10^{12}) tera
G (10^{9}) giga
M (10^{6}) mega
k (10^{3}) kilo
m (10^{-3}) milli
(10^{-6}) micro
n (10^{-9}) nano
p (10^{-12}) pico
f (10^{-15}) fempto
a (10^{-18}) atto

Numbers with SI scale factors are commonly used in science and engineering to represent physical quantities because it is easy to read and write numbers both large and small. For example, the distance between the atoms in a silicon lattice is roughly 230 pm whereas the distance to the sun is about 150 Gm. Unfortunately, computers do not normally use SI scale factors. Instead, they use E-notation. The two distances would be written as 2.3e-10 m and 1.5e+11 m. Virtually all computer languages such as Python both read and write numbers in E-notation, but none naturally read or

write numbers that use SI scale factors, even though SI is an international standard that has been in place for over 50 years and is widely used.

QuantiPhy is an attempt to address both of these deficiencies. It allows quantities to be represented with a single object that allows the complete quantity to be easily read or written as a single unit. It also naturally supports SI scale factors. As such, *QuantiPhy* allows computers to communicate more naturally with humans, particularly scientists and engineers.

7.1.2 Quantities

QuantiPhy is a library that adds support to Python for both reading and writing numbers with SI scale factors and units. The primary working construct for *QuantiPhy* is `Quantity`, which is a class whose objects hold the number and units that are used to represent a physical quantity. For example, to create a quantity from a string you can use:

```
>>> from quantiphy import Quantity

>>> distance_to_sun = Quantity('150 Gm')
>>> distance_to_sun.real
150000000000.0

>>> distance_to_sun.units
'm'

>>> print(distance_to_sun)
150 Gm
```

Now *distance_to_sun* contains an object with two values, the number 150000000000.0 and the units 'm'. The 'G' was interpreted as the *giga* scale factor, which scales 150 by 10^9.

It is worth considering the alternative for a moment:

```
>>> d_sol = float('150000000000.0')
>>> print(f'{d_sol} m')
150000000000.0 m
```

Ignoring the difficulty in writing and reading the number, there is another important difference. The units are placed in the print statement and not kept with the number. This makes the value ambiguous, it clutters the print statement, and it introduces a vulnerability. When coming back and refactoring your code after some time has passed, you might change the units of the number and forget to change the units in the print statement. This is particularly likely if the number is defined far from where it is printed. The result is that erroneous results are printed and is always a risk when two related pieces of information are specified far from one another. *QuantiPhy* addresses this issue by binding the value and the units into one object.

`Quantity` is a subclass of float, and so *distance_to_sun* can be used just like any real number. For example, you can convert the distance to miles using:

```
>>> distance_in_miles = distance_to_sun / 1609.34
>>> print(distance_in_miles)
93205910.49747102
```

When printed or converted to strings quantities naturally use SI scale factors. For example, you can clean up that distance in miles using:

```
>>> distance_in_miles = Quantity(distance_to_sun / 1609.34, 'miles')
>>> print(distance_in_miles)
93.206 Mmiles
```

However, you need not explicitly do the conversion yourself. *QuantiPhy* provides many of the most common conversions for you:

```
>>> distance_in_miles = Quantity(distance_to_sun, scale='miles')
>>> print(distance_in_miles)
93.206 Mmiles
```

Specifying Quantities

Normally, creating a *Quantity* takes one or two arguments. The first is taken to be the value, and the second, if given, is taken to be the model, which is a source of default values.

The first argument: the value

The value may be given as a float, as a string, or as a quantity. The string may be the name of a known constant or it may represent a number. If the string represents a number, it may be in floating point notation (1200.0), in E-notation (ex: 1.2e+3), or use SI scale factors (1.2k). It may also include the units. And like Python in general, the numbers may include underscores to make them easier to read (they are ignored). For example, any of the following ways can be used to specify 1ns:

```
>>> period = Quantity(1e-9, 's')
>>> print(period)
1 ns

>>> period = Quantity('0.000_000_001 s')
>>> print(period)
1 ns

>>> period = Quantity('1e-9s')
>>> print(period)
1 ns

>>> period = Quantity('1ns')
>>> print(period)
1 ns

>>> period2 = Quantity(period)
>>> print(period2)
1 ns
```

If given as a string, the value may also be the name of a known *constant*:

```
>>> k = Quantity('k')
>>> q = Quantity('q')
>>> print(k, q, sep='\n')
13.806e-24 J/K
160.22e-21 C
```

The following constants are pre-defined: h, , k, q, c, $0°C$, $_0$, $_0$, and Z_0. You may add your own *constants*.

Currency units ($€¥£₩) are a bit different than other units in that they are placed at the front of the quantity.

```
>>> print(Quantity('$11_200_000'))
$11.2M

>>> print(Quantity(11.2e6, '$'))
$11.2M
```

When using currency units, if the number has a sign, it should precede the units:

```
>>> print(Quantity('-$11_200_000'))
-$11.2M

>>> print(Quantity(-11.2e6, '$'))
-$11.2M
```

When given as a string, the number may use any of the following scale factors (though you can use the *input_sf* preference to prune this list if desired):

Q (10^{30}) quetta
R (10^{27}) ronna
Y (10^{24}) yotta
Z (10^{21}) zetta
E (10^{18}) exa
P (10^{15}) peta
T (10^{12}) tera
G (10^{9}) giga
M (10^{6}) mega
k (10^{3}) kilo
_ (1)
c (10^{-2}) centi
m (10^{-3}) milli
u (10^{-6}) micro (ASCII)
µ (10^{-6}) micro (unicode micro)
(10^{-6}) micro (unicode Greek mu)
n (10^{-9}) nano
p (10^{-12}) pico
f (10^{-15}) fempto
a (10^{-18}) atto
z (10^{-21}) zepto
y (10^{-24}) yocto
r (10^{-27}) ronto
q (10^{-30}) quecto

In addition, the units must start with a letter or any of these characters: `°ÅΩ℧¢$€¥£₩₹%`, and may be followed by those characters (except %) or digits or any of these characters: `-^/().⁰¹²³⁴⁵⁶⁷⁸⁹`. Thus, any of the following would be accepted as units: `Ohms`, `V/A`, `J-s`, `m/s^2`, `H/(m-s)`, , `%`, `m·s²`, `V/Hz`.

When specifying the value as a string you may also give a name and description, and if you do they become available as the attributes *name* and *desc*. This conversion is under the control of the *assign_rec* preference. The default version of *assign_rec* accepts either '=' or ':' to separate the name from the value, and either '—', '–', '#', or '//' to separate the value from the description if a description is given. Thus, by default *QuantiPhy* recognizes specifications of the following forms:

```
<name> = <value>
<name> = <value> - <description>
<name> = <value> -- <description>
<name> = <value> # <description>
<name> = <value> // <description>
<name>: <value>
<name>: <value> - <description>
<name>: <value> -- <description>
<name>: <value> # <description>
<name>: <value> // <description>
```

For example:

```
>>> period = Quantity('Tclk = 10ns -- clock period')
>>> print(f'{period.name} = {period}  # {period.desc}')
Tclk = 10 ns  # clock period
```

The second argument: the model

If you only specify a real number for the value, then the units, name, and description do not get values. Even if given as a string or quantity, the value may not contain these extra attributes. This is where the second argument, the model, helps. It may be another quantity or it may be a string. Any attributes that are not provided by the first argument are taken from the second if available. If the second argument is a string, it is split. If it contains one value, that value is taken to be the units, if it contains two, those values are taken to be the name and units, and it it contains more than two, the remaining values are taken to be the description. If the model is a quantity, only the units are inherited. For example:

```
>>> out_period = Quantity(10*period, period)
>>> print(out_period)
100 ns

>>> freq = Quantity(100e6, 'Hz')
>>> print(freq)
100 MHz

>>> freq = Quantity(100e6, 'Fin Hz')
>>> print(f'{freq.name} = {freq}')
Fin = 100 MHz

>>> freq = Quantity(100e6, 'Fin Hz input frequency')
>>> print(f'{freq.name} = {freq} - {freq.desc}')
Fin = 100 MHz - input frequency
```

If the model contains units, those units are only used if the value does not have units. The same is true for the description. For example:

```
>>> h = Quantity('18in', 'm')
>>> print(h)
18 in
```

The remaining arguments

Any arguments beyond the first two must be given as named arguments.

If you need to override the name, units or the description given in either the value or the model, you can do so by specifying them with corresponding named arguments. For example:

```
>>> out_period = Quantity(
...     10*period, period, name='output period',
...     desc='period at output of frequency divider'
... )
>>> print(f'{out_period.name} = {out_period} - {out_period.desc}')
output period = 100 ns - period at output of frequency divider
```

In this the value is `10*period`, which is a float and so has no name, units, or description attributes, but the model is `period` that has all three attributes, but the name name and description, coming from a quantity, are ignored. Instead, they are specified explicitly using the *name* and *desc* arguments.

Specifying *binary* as *True* allows you to use the binary scale factors. The binary scale factors are *Ki*, *Mi*, *Gi*, *Ti*, *Pi*, *Ei*, *Zi*, and *Yi*. Unlike the normal scale factors, you cannot use a lower case *k* in *Ki*. Also, *input_sf* is ignored. The normal recognizers are used if none of the binary scale factors are found.

```
>>> bytes = Quantity('1 KiB', binary=True)
>>> print(bytes)
1.024 kB
```

You can also specify *scale* and *ignore_sf* as named arguments. *scale* allows you to scale the value or convert it to different units. It is described *in a bit*. *ignore_sf* indicates that any scale factors should be ignored. This is *one way* of handling units whose name starts with a scale factor character. For example:

```
>>> x = Quantity('1m')                                  # unitless value
>>> print(x, x.real, x.units, sep=', ')
1m, 0.001,

>>> l = Quantity('1m', ignore_sf=True)                  # length in meters
>>> print(l, l.real, l.units, sep=', ')
1 m, 1.0, m

>>> d = Quantity('1m', units = 'mile', ignore_sf=True)  # distance in miles
>>> print(d, d.real, d.units, sep=', ')
1 mile, 1.0, mile

>>> t = Quantity('1m', units = 'min', ignore_sf=True)   # duration in minutes
>>> print(t, t.real, t.units, sep=', ')
1 min, 1.0, min
```

Finally, you can also specify conversion parameters using *params*. These values are ignored by *QuantiPhy* except that they are made available to any *UnitConversion* conversion functions as a way of implementing parametrized conversions.

Quantity attributes

You can overwrite *Quantity* attributes to override the units, name, or description.

```
>>> out_period = Quantity(10*period)
>>> out_period.units = 's'
>>> out_period.name = 'output period'
>>> out_period.desc = 'period at output of frequency divider'
>>> print(f'{out_period.name} = {out_period} - {out_period.desc}')
output period = 100 ns - period at output of frequency divider
```

In addition, you can also override the preferences with attributes:

```
>>> out_period.spacer = ''
>>> print(out_period)
100ns
```

Scaling When Creating a Quantity

Quantities tend to be used primarily when reading and writing numbers, and less often when processing numbers. Often data comes in an undesirable form. For example, imagine data that has been normalized to kilograms but the numbers themselves have neither units or scale factors. *QuantiPhy* allows you to scale the number and assign the units when creating the quantity:

```
>>> mass = Quantity('2.529', scale=1000, units='g')
>>> print(mass)
2.529 kg
```

In this case the value is given in kilograms, and is converted to the base units of grams by multiplying the given value by 1000. You always want to convert to base units (units with no scale factor) when creating a *Quantity*. This can also be expressed as follows:

```
>>> mass = Quantity('2.529', scale=(1000, 'g'))
>>> print(mass)
2.529 kg
```

You can also specify a function to do the conversion, which is helpful when the conversion is not linear:

```
>>> def from_dB(value, units=''):
...     return 10**(value/20), value.units[2:]

>>> Quantity('-100 dBV', scale=from_dB)
Quantity('10 uV')
```

Note: Since version 2.18 the first argument, in this case *value*, is guaranteed to be a *Quantity* that contains both the units and any parameters needed during the conversion. As such, the second argument, *units*, is not longer needed and will eventually be removed.

The conversion can also often occur if you simply state the units you wish the quantity to have:

```
>>> Tboil = Quantity('212 °F', scale='K')
>>> print(Tboil)
373.15 K
```

or if you employ a subclass of *Quantity* that has units:

```
>>> class Kelvin(Quantity):
...     units = 'K'

>>> Tboil = Kelvin('212 °F')
>>> print(Tboil)
373.15 K
```

This assumes that the initial value is specified with units. If not, you need to provide them for these mechanisms to work.

```
>>> Tboil = Quantity('212', '°F', scale='K')
>>> print(Tboil)
373.15 K
```

To do this conversion, *QuantiPhy* examines the given units (°F) and the desired units (K) and chooses the appropriate converter. No scaling is done if the given units are equivalent as the desired units. Thus you can use the scaling mechanism to convert a collection of data with mixed units to values with consistent units. For example:

```
>>> weights = '''
...     240 lbs
...     230 lb
...     100 kg
...     210
... '''.strip().split('\n')
>>> for weight in weights:
...     w = Quantity(weight, 'lb', scale='lb')
...     print(w)
240 lb
230 lb
220.46 lb
210 lb
```

To perform these conversions *QuantiPhy* uses predefined relationships between pairs of units. These relationships are defined using *Unit Converters*.

When using unit conversions it is important to only convert to units without scale factors when creating a quantity. For example, it is better to convert to 'g' rather than 'kg'. Otherwise, if the desired units used when creating a quantity includes a scale factor, it is easy to end up with two scale factors when converting the number to a string (ex: 1 mkg or one milli-kilo-gram).

Here is another example that uses quantity scaling. Imagine that a table is being read that gives temperature versus time, but the temperature is given in °F and the time is given in minutes and neither are given with units. Assume that for the purpose of later analysis it is desirable for the values be converted to the more natural units of Kelvin and seconds:

```
>>> rawdata = '0 450, 10 400, 20 360'
>>> data = []
>>> for pair in rawdata.split(','):
```

(continues on next page)

(continued from previous page)

```
...     time, temp = pair.split()
...     time = Quantity(time, 'min', scale='s')
...     temp = Quantity(temp, '°F', scale='K')
...     data += [(time, temp)]

>>> for time, temp in data:
...     print(f'{time:9q} {temp:9q}')
      0 s  505.37 K
    600 s  477.59 K
   1.2 ks  455.37 K
```

Creating a Quantity by Scaling an Existing Quantity

The *Quantity.scale()* method scales the value of a quantity and then uses the new value to create a new Quantity. For example:

```
>>> import math

>>> h_line = Quantity('1420.405751786 MHz')
>>> sagan = h_line.scale(math.pi)
>>> sagan2 = sagan.scale(2)
>>> print(sagan, sagan2, sep='\n')
4.4623 GHz
8.9247 GHz

>>> print(repr(h_line))
Quantity('1.420405751786 GHz')

>>> print(repr(sagan))
Quantity('4.462336274928 GHz')
```

Any value that can be passed to the *scale* argument for *Quantity* or *Quantity.render()* can be passed to the *scale* method. Specifically, the following types are accepted:

float or Quantity
: The argument scales the underlying value (a new quantity is returned whose value equals the underlying quantity multiplied by scale). In this case the scale is assumed unitless (any units are ignored) and so the units of the new quantity are the same as those of the underlying quantity.

tuple
: The argument consists of two values. Tthe first value, a float, is treated as a scale factor. The the second value, a string, is taken to be the units of the new quantity.

function
: The function takes two arguments, the value to be scaled and its units. The value is guaranteed to be a Quantity that includes the units, so the second argument is redundant and will eventually be deprecated. The function returns two values, the value and units of the new value.

string
: The argument is taken to the be desired units. This value along with the units of the underlying quantity are used to select a known unit conversion, which is applied to create the new value.

```
>>> Tboil_C = Tboil.scale('C')
>>> print(Tboil_C)
100 C
```

Creating a Quantity by Adding to an Existing Quantity

The *Quantity.add()* method adds a contribution to the value of a quantity and then uses the sum to create a new Quantity. For example:

```
>>> import math

>>> total = Quantity(0, '$')
>>> for contribution in ['1.23', '4.56', '7.89']:
...     total = total.add(contribution)
>>> print(total)
$13.68
```

The argument to *add* can be a quantity, a real number, or a string.

When adding quantities, the units of the quantity should match. You can enforce this by adding *check_units=True*. If the dimension of your quantities match but not the units, you can often use *Quantity.scale()* to get the units right:

```
>>> m1 = Quantity('1kg')
>>> m2 = Quantity('1lb')
>>> m3 = m1.add(m2.scale('g'), check_units=True)
>>> print(m3)
1.4536 kg
```

Accessing Quantity Values

There are a variety of ways of accessing the value of a quantity. If you are just interested in its numeric value, you access it with:

```
>>> h_line.real
1420405751.786

>>> float(h_line)
1420405751.786
```

Or you can simply use a quantity in the same way that you would use any real number, meaning that you can use it in expressions and it evaluates to its numeric value:

```
>>> second_sagan_freq = 2 * math.pi * h_line
>>> print(second_sagan_freq)
8924672549.85517

>>> sagan2 = Quantity(second_sagan_freq, h_line)
>>> print(sagan2)
8.9247 GHz

>>> type(h_line)
```

(continues on next page)

(continued from previous page)

```
<class 'quantiphy.quantiphy.Quantity'>

>>> type(second_sagan_freq)
<class 'float'>

>>> type(sagan2)
<class 'quantiphy.quantiphy.Quantity'>
```

Notice that when performing arithmetic operations on quantities the units are completely ignored and do not propagate in any way to the newly computed result.

If you are interested in the units of a quantity, you can use:

```
>>> h_line.units
'Hz'
```

Or you can access both the value and the units, either as a tuple or in a string:

```
>>> h_line.as_tuple()
(1420405751.786, 'Hz')

>>> str(h_line)
'1.4204 GHz'
```

SI scale factors are used by default when converting numbers to strings. The following scale factors could be used: `QRYZEPTGMkc%munpfazyrq`, though by default % is treated as a unit rather than a scale factor. You need to activate % in *input_sf* for it to be treated as a scale factor.

Only the scale factors listed in the *output_sf* preference are actually used, and by default that is set to `TGMkmunpfa`, which avoids the more uncommon scale factors. You can set *output_sf* to *Quantity.all_sf* to output all known scale factors except `c` or `%`, which are never used in output.

The *Quantity.render()* method allows you to control the process of converting a quantity to a string. For example:

```
>>> h_line.render()
'1.4204 GHz'

>>> h_line.render(form='eng')
'1.4204e9 Hz'

>>> h_line.render(show_units=False)
'1.4204G'

>>> h_line.render(form='eng', show_units=False)
'1.4204e9'

>>> h_line.render(prec=6)
'1.420406 GHz'

>>> h_line.render(form='fixed', prec=2)
'1420405751.79 Hz'

>>> bytes.render(form='binary')
'1 KiB'
```

(continues on next page)

(continued from previous page)

```
>>> k.render(negligible=1e-12)
'0 J/K'
```

show_label allows you to display the name and description of the quantity when rendering. If *show_label* is *False*, the quantity is not labeled with the name or description. Otherwise the quantity is labeled under the control of the *show_label* value and the *show_desc*, *label_fmt* and *label_fmt_full* preferences (described further in *Preferences* and *Quantity.set_prefs()*). If *show_label* is 'a' (for abbreviated) or if the quantity has no description, *label_fmt* is used to label the quantity with its name. If *show_label* is 'f' (for full), *label_fmt_full* is used to label the quantity with its name and description. Otherwise *label_fmt_full* is used if *show_desc* is True and *label_fmt* otherwise.

```
>>> freq.render(show_label=True)
'Fin = 100 MHz'

>>> freq.render(show_label='f')
'Fin = 100 MHz - input frequency'

>>> Quantity.set_prefs(show_desc=True)
>>> freq.render(show_label=True)
'Fin = 100 MHz - input frequency'

>>> freq.render(show_label='a')
'Fin = 100 MHz'
```

You can also access the full precision of the quantity:

```
>>> h_line.render(prec='full')
'1.420405751786 GHz'

>>> h_line.render(form='eng', prec='full')
'1.420405751786e9 Hz'
```

Full precision implies whatever precision was used when specifying the quantity if it was specified as a string and if the *keep_components* preference is True. Otherwise a fixed number of digits, specified in the *full_prec* preference, is used (default=12). Generally one uses 'full' when generating output that is intended to be read by a machine without loss of precision.

An alternative to *render* is *Quantity.fixed()*. It converts the quantity to a string in fixed-point format:

```
>>> total = Quantity('$11.2M')
>>> print(total.fixed(prec=2, show_commas=True, strip_zeros=False))
$11,200,000.00
```

You can also use *Quantity.render()* to produce a fixed format, but it does not support all of the options available with *fixed*:

```
>>> print(total.render(form='fixed', prec=2))
$11200000
```

Another alternative to *render* is *Quantity.binary()*. It converts the quantity to a string that uses binary scale factors:

```
>>> mem = Quantity(17_179_869_184, 'B', name='physical memory')
>>> print(mem.binary())
16 GiB
```

Alternatively you can also use *render* to render strings with binary prefixes:

```
>>> print(mem.render(form='binary'))
16 GiB
```

Scaling When Rendering a Quantity

Once it comes time to output quantities from your program, you may again may be constrained in the way the numbers must be presented. *QuantiPhy* also allows you to scale the values as you render them to strings. In this case, the value of the quantity itself remains unchanged. For example, imagine having a quantity in grams and wanting to present it in either kilograms or in pounds:

```
>>> m = Quantity('2529 g')
>>> print("mass (kg): {}".format(m.render(show_units=False, scale=0.001)))
mass (kg): 2.529

>>> print(m.render(scale=(0.0022046, 'lb'), form='fixed'))
5.5754 lb
```

As before, functions can also be used to do the conversion. Here is an example where that comes in handy: a logarithmic conversion to dBV is performed.

```
>>> import math
>>> def to_dB(value, units=''):
...     return 20*math.log10(value), 'dB'+value.units

>>> T = Quantity('100mV')
>>> print(T.render(scale=to_dB))
-20 dBV
```

Note: Since version 2.18 the first argument, in this case *value*, is guaranteed to be a *Quantity* that contains both the units and any parameters needed during the conversion. As such, the second argument, *units*, is not longer needed and will eventually be removed.

Finally, you can also use either the built-in converters or the converters you created to do the conversion simply based on the units:

```
>>> print(m.render(scale='lb'))
5.5755 lb
```

In an earlier example the units of time and temperature data were converted to normal SI units. Presumably this makes processing easier. Now, when producing the output, the units can be converted back to the original units if desired:

```
>>> for time, temp in data:
...     print("{:<7} {}".format(time.render(scale='min'), temp.render(scale='°F')))
0 min   450 °F
10 min  400 °F
20 min  360 °F
```

String Formatting

Quantities can be passed into the string *format* method:

```
>>> print('{}'.format(h_line))
1.4204 GHz

>>> print('{:s}'.format(h_line))
1.4204 GHz
```

In these cases the preferences for SI scale factors, units, and precision are honored.

Specifying the format

You can override the precision as part of the format specification

```
>>> print('{:.6}'.format(h_line))
1.420406 GHz
```

You can also specify the width and alignment. *Quantiphy* follows the Python convention of right justifying numbers by default.

```
>>> print('«{:16.6}»'.format(h_line))
«    1.420406 GHz»

>>> print('«{:<16.6}»'.format(h_line))
«1.420406 GHz    »

>>> print('«{:>16.6}»'.format(h_line))
«    1.420406 GHz»

>>> print('«{:^16.6}»'.format(h_line))
«  1.420406 GHz  »
```

The general form of the format specifiers supported by quantities is:

```
format_spec ::=  [align][#][width][,][.precision][type][scale]
```

align specifies the alignment using one of the following characters:

Align	Meaning
>	Right justification.
<	Left justification.
^	Center justification.

The hash (#) is a literal hash that when present indicates that trailing zeros and radix should not be stripped from the fractional part of the number.

width is a literal integer that specifies the minimum width of the string.

The comma (,) is a literal comma that when present indicates that commas should be added to the whole part of the mantissa, every three digits.

precision is a literal integer that specifies the precision.

And finally, *type* specifies which form should be used when formatting the value. The choices include:

Type	Meaning
None	Use default formatting options.
s	Use default formatting options.
q	Format using SI scale factors and show the units.
r	Format using SI scale factors but do not show the units.
p	Format using fixed-point notation and show the units.
e	Format using exponent notation but do not show the units.
f	Format using fixed-point notation but do not show the units.
b	Format using binary prefixes while showing the units.
g	Format using fixed-point or exponential notation, whichever is shorter, but do not show the units.
u	Only include the units.
n	Only include the name.
d	Only include the description.

You can capitalize any of the format characters that output the value of the quantity (any of 'sqrpefg', but not 'und'). If you do, the label will also be included.

These format specifiers are generally included in format strings. However, in addition, *Quantitphy* provides the *Quantity.format()* method that converts a quantity to a string based on a naked format string. For example:

```
>>> print(h_line.format('.6q'))
1.420406 GHz
```

Any format specification that is not recognized by *QuantiPhy* is simply passed on to the underlying float. For example:

```
>>> print(f'TOTAL: {total:+#,.2f}')
TOTAL: +11,200,000.00

>>> with Quantity.prefs(input_sf='%'):
...     growth = Quantity('23.7%')
>>> print(f'growth = {growth:.0%}')
growth = 24%
```

Examples

Here is an example of these format types:

```
>>> h_line = Quantity('f = 1420.405751786 MHz - hydrogen line')
>>> for f in 'sSpPqQrRbBeEfFgGund':
...     print(f + ':', h_line.format(f))
s: 1.4204 GHz
S: f = 1.4204 GHz - hydrogen line
p: 1420405751.786 Hz
P: f = 1420405751.786 Hz - hydrogen line
q: 1.4204 GHz
Q: f = 1.4204 GHz - hydrogen line
r: 1.4204G
R: f = 1.4204G - hydrogen line
b: 1.3229 GiHz
B: f = 1.3229 GiHz - hydrogen line
```

(continues on next page)

(continued from previous page)

```
e: 1.4204e+09
E: f = 1.4204e+09 - hydrogen line
f: 1420405751.786
F: f = 1420405751.786 - hydrogen line
g: 1.4204e+09
G: f = 1.4204e+09 - hydrogen line
u: Hz
n: f
d: hydrogen line
```

The 'q' type specifier is used to explicitly indicate that both the number and the units are desired and that SI scale factors should be used, regardless of the current preferences.

```
>>> print('{:.6q}'.format(h_line))
1.420406 GHz
```

Alternately, 'r' can be used to indicate just the number represented using SI scale factors is desired, and the units should not be included.

```
>>> print('{:r}'.format(h_line))
1.4204G
```

The opposite can be achieved using 'p', which includes the units without SI scale factors:

```
>>> print('{:p}'.format(h_line))
1420405751.786 Hz
```

The 'p' format is often used with '#' to format currency values:

```
>>> print('{:#.2p}'.format(total))
$11200000.00

>>> print('{:#,.2p}'.format(total))
$11,200,000.00
```

The 'b' format is used to render number with binary scale factors:

```
>>> print('{:b}'.format(mem))
16 GiB

>>> print('{:B}'.format(mem))
physical memory = 16 GiB
```

You can also use the traditional floating point format type specifiers:

```
>>> print('{:f}'.format(h_line))
1420405751.786

>>> print('{:e}'.format(h_line))
1.4204e+09

>>> print('{:g}'.format(h_line))
1.4204e+09
```

Use 'u' to indicate that only the units are desired:

```
>>> print('{:u}'.format(h_line))
Hz
```

Access the name or description of the quantity using 'n' and 'd'.

```
>>> print('{:n}'.format(freq))
Fin

>>> print('{:d}'.format(freq))
input frequency
```

Using the upper case versions of the format codes that print the numerical value of the quantity (SQRFEG) indicates that the quantity should be labeled with its name and perhaps its description (as if the *show_label* preference were set). They are under the control of the *show_desc*, *label_fmt* and *label_fmt_full* preferences (described further in *Preferences* and *Quantity.set_prefs()*).

If *show_desc* is False or the quantity does not have a description, then *label_fmt* is used to add the labeling.

```
>>> Quantity.set_prefs(show_desc=False)
>>> trise = Quantity('10ns', name='trise')

>>> print('{:S}'.format(trise))
trise = 10 ns

>>> print('{:Q}'.format(trise))
trise = 10 ns

>>> print('{:R}'.format(trise))
trise = 10n

>>> print('{:F}'.format(trise))
trise = 0

>>> print('{:E}'.format(trise))
trise = 1e-08

>>> print('{:G}'.format(trise))
trise = 1e-08

>>> print('{0:n} = {0:q} ({0:d})'.format(freq))
Fin = 100 MHz (input frequency)

>>> print('{:S}'.format(freq))
Fin = 100 MHz
```

If *show_desc* is True and the quantity has a description, then *label_fmt_full* is used if the quantity has a description.

```
>>> Quantity.set_prefs(show_desc=True)

>>> print('{:S}'.format(trise))
trise = 10 ns
```

(continues on next page)

(continued from previous page)

```
>>> print('{:S}'.format(freq))
Fin = 100 MHz - input frequency
```

Scaling while formatting

Finally, you can add units after the format code, which causes the number to be scaled to those units if the transformation represents a known unit conversion. In this case the format code must be specified (use 's' rather than '').

```
>>> Tboil = Quantity('Boiling point = 100 °C')
>>> print('{:S°F}'.format(Tboil))
Boiling point = 212 °F

>>> eff_channel_length = Quantity('leff = 14nm')
>>> print(f'{eff_channel_length:SÅ}')
leff = 140 Å

>>> print(f'{mem:bb}')
128 Gib
```

This feature can be used to simplify the conversion of the time and temperature information back into the original units:

```
>>> for time, temp in data:
...     print(f'{time:<7smin} {temp:s°F}')
0 min   450 °F
10 min  400 °F
20 min  360 °F
```

You can add a scale factor to the units, in which case the number will be scaled accordingly:

```
>>> for p in range(1, 5):
...     bytes = Quantity(256**p, 'B')
...     print(f"An {8*p} bit bus addresses {bytes:,pkB}.")
An 8 bit bus addresses 0.256 kB.
An 16 bit bus addresses 65.536 kB.
An 24 bit bus addresses 16,777.216 kB.
An 32 bit bus addresses 4,294,967.296 kB.
```

Generally you should only specify base units when using a format that renders with scale factors as otherwise you could see two scale factors on the same number. For example, if the q format was used in the above example, the last address space would be rendered as 4.295 MkB.

7.1.3 Ambiguity of Scale Factors and Units

By default, *QuantiPhy* treats both the scale factor and the units as being optional. With the scale factor being optional, the meaning of some specifications can be ambiguous. For example, '1m' may represent 1 milli or it may represent 1 meter. Similarly, '1meter' my represent 1 meter or 1 milli-eter. In this case *QuantiPhy* gives preference to the scale factor, so '1m' normally converts to 1e-3. To allow you to avoid this ambiguity, *QuantiPhy* accepts '_' as the unity scale factor. In this way '1_m' is unambiguously 1 meter. You can instruct *QuantiPhy* to output '_' as the unity scale factor by specifying the *unity_sf* argument to *Quantity.set_prefs()*:

```
>>> Quantity.set_prefs(unity_sf='_', spacer='')
>>> l = Quantity(1, 'm')
>>> print(l)
1_m
```

This is often a good way to go if you are outputting numbers intended to be read unambiguously or by both people and machines.

If you need to interpret numbers that have units and are known not to have scale factors, you can specify the *ignore_sf* preference:

```
>>> Quantity.set_prefs(ignore_sf=True, unity_sf='', spacer=' ')
>>> l = Quantity('1000m')
>>> l.as_tuple()
(1000.0, 'm')

>>> print(l)
1 km

>>> Quantity.set_prefs(ignore_sf=False)
>>> l = Quantity('1000m')
>>> l.as_tuple()
(1.0, '')
```

If there are scale factors that you know you will never use, you can instruct *QuantiPhy* to interpret a specific set and ignore the rest using the *input_sf* preference.

```
>>> Quantity.set_prefs(input_sf='GMk')
>>> l = Quantity('1000m')
>>> l.as_tuple()
(1000.0, 'm')

>>> print(l)
1 km
```

Specifying *input_sf=None* causes *QuantiPhy* to again accept all known scale factors.

```
>>> Quantity.set_prefs(input_sf=None)
>>> l = Quantity('1000m')
>>> l.as_tuple()
(1.0, '')
```

Alternatively, you can specify the units you wish to use whose leading character is a scale factor. Once known, these units no longer confuse *QuantiPhy*. These units can be specified as a list or as a string. If specified as a string the string is split to form the list. Specifying the known units replaces any existing known units.

```
>>> d1 = Quantity('1 au')     # astronomical unit
>>> d2 = Quantity('1000 pc')  # parsec
>>> p = Quantity('138 Pa')    # Pascal
>>> print(d1.render(form='eng'), d2, p, sep='\n')
1e-18 u
1 nc
138e15 a
```

(continues on next page)

(continued from previous page)

```
>>> Quantity.set_prefs(known_units='au pc Pa')
>>> d1 = Quantity('1 au')
>>> d2 = Quantity('1000 pc')
>>> p = Quantity('138 Pa')
>>> print(d1.render(form='eng'), d2, p, sep='\n')
1 au
1 kpc
138 Pa
```

This same issue comes up for temperature quantities when given in Kelvin. There are again several ways to handle this. First you can specify the acceptable input scale factors leaving out 'K', ex. *input_sf* = 'TGMkmunpfa', or:

```
>>> Quantity.set_prefs(input_sf=Quantity.get_pref('input_sf').replace('K', ''))
>>> temp = Quantity('100K')
>>> print(temp.as_tuple())
(100.0, 'K')

>>> temp = Quantity('100k')
>>> print(temp.as_tuple())
(100000.0, '')

>>> temp = Quantity('100k', 'K')
>>> print(temp.as_tuple())
(100000.0, 'K')
```

Alternatively, you can specify 'K' as one of the known units. Finally, if you know exactly when you will be converting a temperature to a quantity, you can specify *ignore_sf* for that specific conversion. The effect is the same either way, 'K' is interpreted as a unit rather than a scale factor.

The same techniques would be used to handle volumes in cubic centimeters:

```
>>> vol = Quantity('10 cc')
>>> print(vol.as_tuple())
(0.1, 'c')
```

```
>>> with Quantity.prefs(input_sf=Quantity.get_pref('input_sf').replace('c', '')):
...     vol = Quantity('10 cc')
>>> print(vol.as_tuple())
(10.0, 'cc')
```

```
>>> with Quantity.prefs(known_units='cc'):
...     vol = Quantity('100 cc')
>>> print(vol.as_tuple())
(100.0, 'cc')
```

Percentages are a special case. *QuantiPhy* can treat the % character as either a unit or a scale factor (0.01). By default it is treated as a unit:

```
>>> tolerance = Quantity('10%')
>>> change = Quantity('10%')
>>> print(tolerance.as_tuple(), change.as_tuple(),)
(10.0, '%') (10.0, '%')
```

If, however, you add % as a known scale factor, it then acts as a scale factor.

```
>>> with Quantity.prefs(input_sf = Quantity.get_pref('input_sf') + '%'):
...     tolerance = Quantity('10%')
...     change = Quantity('10%')
...     print(tolerance.as_tuple(), change.as_tuple(),)
(0.1, '') (0.1, '')
```

In general you cannot simply add to the list of known scale factors. The % character is an exception as *QuantiPhy* knows about it but disables it by default.

7.1.4 Subclassing Quantity

You can subclass *Quantity* to make it easier to create a particular type of quantity, or to create quantities with particular qualities. The following example demonstrates both. It creates a subclass for dollars that both sets the units and the display preferences. Any *Quantity* preference (see *Quantity.set_prefs()*) may be given as an attribute. Display preferences for currencies are often very different from what you would want from physical quantities:

```
>>> class Dollars(Quantity):
...     units = '$'
...     form = 'fixed'
...     prec = 2
...     strip_zeros = False
...     show_commas = True

>>> cost = Dollars(100_000)
>>> print(cost)
$100,000.00
```

This example creates a special class for bytes.

```
>>> class Bytes(Quantity):
...     units = 'B'
...     form = 'binary'
...     accept_binary = True

>>> memory = Bytes('64KiB')
>>> print(memory)
64 KiB
```

Here, two classes are created for voltage and current, each with their own perspective on what values should be considered negligible.

```
>>> class Voltage(Quantity):
...     units = 'V'
...     negligible = 1e-6

>>> class Current(Quantity):
...     units = 'A'
...     negligible = 1e-12

>>> Vout = Voltage(1e-9)
>>> Ileak = Current(1e-9)
```

(continues on next page)

(continued from previous page)

```
>>> print(f"Vout = {Vout}, Ileak = {Ileak}.")
Vout = 0 V, Ileak = 1 nA.
```

Lastly, this example creates a special class for temperatures. It disallows use of 'K' as a scale factor to avoid confusion with Kelvin units.

```
>>> class Temperature(Quantity):
...     units = 'K'
...     input_sf = Quantity.get_pref('input_sf').replace('K', '')

>>> Tcore = Temperature('15M')
>>> Tphoto = Temperature('5.3k')
>>> Tcmb = Temperature('3.18K')
>>> print(Tcore, Tphoto, Tcmb, sep='\n')
15 MK
5.3 kK
3.18 K
```

Scaling with Subclasses

Special scaling rules come into play if the *units* attribute is present on a *Quantity* class. In such a case you can specify the class as an argument to a scaling operation. For example:

```
>>> class Grams(Quantity):
...     units = 'g'

>>> class Pounds(Quantity):
...     units = 'lbs'

>>> wt = Pounds(10)
>>> mass = wt.scale(Grams)

>>> print(mass, repr(mass), sep='\n')
4.5359 kg
Grams('4.5359237 kg')

>>> print(wt.render(scale=Grams))
4.5359 kg
```

Notice that use of *Grams* with the *Quantity.scale()* method resulted in a return value of type *Grams*. This does not naturally occur if you scale using scale factors or units:

```
>>> mass = wt.scale('g')
>>> print(mass, repr(mass), sep='\n')
4.5359 kg
Quantity('4.5359237 kg')
```

In this case you can replicate the previous behavior by adding *Grams* as an argument to the conversion:

```
>>> mass = wt.scale('g', cls=Grams)
>>> print(mass, repr(mass), sep='\n')
```

(continues on next page)

(continued from previous page)

```
4.5359 kg
Grams('4.5359237 kg')
```

Scaling Upon Subclass Creation

When creating quantities using a subclass, a conversion automatically occurs if both the subclass and the value have units. The conversion converts the given units to those expected by the class. For example:

```
>>> class Seconds(Quantity):
...     units = 's'

>>> ttl = Seconds('2 days')
>>> print(ttl)
172.8 ks
```

If you also specify a *scale* argument, that conversion occurs before the result is converted to the units of the class:

```
>>> class Days(Quantity):
...     units = 'days'

>>> expires = Days('48 hr', scale='s')
>>> print(expires)
2 days
```

Adding the *scale* argument is handy because *QuantiPhy* does not provide a built-in direct conversion between hours and days. In this case two conversions occur, from hours to seconds, as a result of the scale request, and from seconds to days, to convert to the units expected by the class.

7.1.5 Unit Converters

The *UnitConversion* class defines conversion relationships between pairs of units, which saves you the trouble of having to remember the actual conversion factors. Once defined, a relationship is available anywhere in *QuantiPhy* where a unit conversion can occur. For example:

```
>>> from quantiphy import Quantity, UnitConversion

>>> m_smoot = UnitConversion('m', 'smoots', 1.7)

>>> length_of_harvard_bridge = Quantity('364.4 smoots')
>>> print(length_of_harvard_bridge.render(scale='m', prec=1))
620 m
```

This is a linear conversion. This unit conversion says, when converting *smoots* to *m*, multiply by 1.7. When going the other way, divide by 1.7.

You can also specify units with a scale factor when scaling a number. For example, you can explicitly direct that the length of the bridge should be output in kilometers using:

```
>>> print(f"{length_of_harvard_bridge:.2pkm}")
0.62 km
```

QuantiPhy* provides a collection of built-in converters for common units:

base units	related units
C °C	K, F °F, R °R
K	C °C, F °F, R °R
m	micron, Å angstrom, mi mile miles, ft feet, in inch inches
g	oz, lb lbs
s	sec second seconds, min minute minutes, hour hours hr, day days
b	B
BTC btc	sat sats ṣ

The conversions can occur between a pair of units, one from the first column and one from the second. They do not occur when both units are only in the second column. So for example, it is possible to convert between *g* and *lbs*, but not between *oz* and *lb*. However, if you notice, the units in the second column are grouped using commas. A set of units within commas are considered equivalent, meaning that there are multiple names for the same underlying unit. For example, *in*, *inch*, and *inches* are all considered equivalent. You can convert between equivalent units even though both are found in either the first or second columns.

UnitConversion supports linear conversions (slope only), affine conversions (slope and intercept) nonlinear conversions, parameterized conversions (conversions with extra parameters) and dynamic conversions (convertions that change over time). Here are some examples:

```
>>> def from_dB(dB):
...     return 10**(dB/20)

>>> def to_dB(v):
...     return 20*math.log10(v)

>>> m_inch = UnitConversion('m', 'in inch inches', 0.0254)  # linear
>>> C_F = UnitConversion('C °C', 'F °F', 5/9, -32*5/9)      # affine
>>> _dB = UnitConversion('', 'dB', from_dB, to_dB)          # nonlinear

>>> print(Quantity('12 in', scale='m'))
304.8 mm

>>> print(Quantity('100 °C', scale='°F'))
212 °F

>>> print(Quantity('100', scale='dB'))
40 dB
```

One thing to be aware of with affine conversions like °C to °F: they are suitable for converting absolute temperatures but not temperature differences. One way around this is to add another conversion specifically for differences:

```
>>> dC_F = UnitConversion('C °C', 'F °F', 5/9)
>>> print(Quantity('100 °C', scale='°F'))
180 °F
```

Notice that the scaling functions used here differ from those described previously in that they only take one argument and return one value. The units are not included in either then argument list or the return value.

Also notice that the return value of *UnitConversion* was not used in the examples above. It is enough to simply create the *UnitConversion* for it to be available to *Quantity*. So, it is normal to not capture the return value of *UnitConversion*. However, there are a few things you can do with the return value. First you can convert it to a string to get a description of the relationship. This is largely used as a sanity check:

```
>>> print(C_F)
C ← 0.5555555555555556*F + -17.778
```

In addition, you can use it to directly perform conversions:

```
>>> temp_F = C_F.convert(0, '°C', '°F')
>>> print(temp_F)
32 °F

>>> temp_C = C_F.convert(32, '°F', '°C')
>>> print(temp_C)
0 °C
```

Finally, you can pre-define multiple conversions between the same pairs of units, and activate the one you currently wish to use. This can be useful with conversions that change over time. For example

```
>>> btc_usd_2017_peak = UnitConversion('USD $', 'BTC ', 19870.62)
>>> btc_usd_2021_peak = UnitConversion('USD $', 'BTC ', 68978.64)

>>> print(Quantity("5 BTC", scale='$'))
$344.89k

>>> btc_usd_2017_peak.activate()
>>> print(Quantity("5 BTC", scale='$'))
$99.353k

>>> btc_usd_2021_peak.activate()
>>> print(Quantity("5 BTC", scale='$'))
$344.89k
```

Defining a conversion between the same pair of units acts to conceal an earlier definition, but the previous definition can be restored using *activate()*.

Parametrized Unit Converters

Occasionally you might encounter conversion that requires one or more extra parameters. For example, to convert between concentration and molarity in solutions requires the atomic weight of the solute. These extra parameters can be passed in when creating a quantity and then are available to the desired conversion. For example:

```
>> @UnitConversion.fixture
>> def from_molarity(M, mw):
..     return M * mw

>> @UnitConversion.fixture
>> def to_molarity(g_L, mw):
..     return g_L / mw

>> mol_conv = UnitConversion('g/L', 'M', from_molarity, to_molarity)

>> KCl_conc = Quantity('1.2 mg/L', params=74.55)
>> print(f"{KCl_conc:qM}")
16.097 uM
```

For more information on defining unit converters, see *UnitConversion*. For more information on parametrized unit converters, see *UnitConversion.fixture()*. For example of real-time dynamic conversions, see *Dynamic Unit Conversions*.

7.1.6 Scale Factor Conversions

In the preceding sections it was shown that you can use the scaling features of *QuantiPhy* to convert between units using only the name of the units. When doing so the relationship between the units must be known, and *UnitConversion* is used to specify the relationship. However, it is also possible to perform simple scale factor conversions without changing the units. This case is specified in a manner similar to a unit conversion, but in this case both the from-units and the to-units are the same, and it is not necessary to define a *UnitConversion*. For example, imagine printing a table of bit-rates where the rates are held in bps but are expected to be displayed in Mbps:

```
>>> rates = [155.52e6, 622.08e6, 2.48832e9, 9.95328e9, 39.81312e9]
>>> rates = [Quantity(r, 'bps') for r in rates]
>>> for r in rates:
...     print(f"{r:>14,.2pMbps}")
   155.52 Mbps
   622.08 Mbps
 2,488.32 Mbps
 9,953.28 Mbps
39,813.12 Mbps
```

You can also do the inverse; convert simple numbers given in Mbps to quantities in bps:

```
>>> rates = [155.52, 622.08, 2488.32, 9953.28, 39813.12]
>>> rates = [Quantity(r, 'Mbps', scale='bps') for r in rates]
>>> for r in rates:
...     print(r.as_tuple())
(155520000.0, 'bps')
(622080000.0, 'bps')
(2488320000.0, 'bps')
(9953280000.0, 'bps')
(39813120000.0, 'bps')
```

7.1.7 Quantity Functions

It is sometimes handy to convert directly to and from real values rather than converting to *Quantity* objects and holding them. Generally it is preferred to key a value and its units together, but as said before, the primary use of *QuantiPhy* is inputting and outputting numbers. If you are not inputting and outputting the same numbers, it may not be worth even the small overhead of a *Quantity* object. In that case, you can use quantity functions to convert directly to and from real values. If you wish to use *QuantiPhy* to convert to a simple float, use *as_real()*. It takes the same arguments as a *Quantity*, but returns a float rather than a *Quantity*:

```
>>> from quantiphy import as_real
>>> print(as_real('10 mL'))
0.01
```

It is common to use *Scale Factor Conversions* to scale the result to the desired size:

```
>>> print(as_real('10 mL', scale='uL'))
10000.0
```

as_tuple() is similar except it returns both the value and the units as a tuple:

```
>>> from quantiphy import as_tuple
>>> print(as_tuple('10 mL'))
(0.01, 'L')

>>> print(as_tuple('10 mL', scale='uL'))
(10000.0, 'uL')
```

Finally, you can use *render()*, *fixed()*, and *binary()* to convert a real value and units into a string. Besides the value and the units, the these functions the same arguments as *Quantity.render()*, *Quantity.fixed()*, and *Quantity.binary()*.

```
>>> from quantiphy import render, fixed, binary
>>> print(render(1e-6, 'L'))
1 uL

>>> print(fixed(1e7, '$', show_commas=True, strip_zeros=False, prec=2))
$10,000,000.00

>>> print(binary(2**32, 'B'))
4 GiB
```

7.1.8 Physical Constants

QuantiPhy has several built-in constants that are available by specifying their name to the *Quantity* class. The following quantities are built in:

Name	MKS value	CGS value	Description
h	6.626070040e-34 J-s	6.626070040e-27 erg-s	Plank's constant
hbar,	1.054571800e-34 J-s	1.054571800e-27 erg-s	Reduced Plank's constant
k	1.38064852e-23 J/K	1.38064852e-16 erg/K	Boltzmann's constant
q	1.6021766208e-19 C	4.80320425e-10 Fr	Elementary charge
c	2.99792458e8 m/s	2.99792458e8 m/s	Speed of light
0C, 0°C	273.15 K	273.15 K	0 Celsius
eps0, $_0$	8.854187817e-12 F/m	—	Permittivity of free space
mu0, $_0$	4e-7 H/m	—	Permeability of free space
Z0, Z_0	376.730313461 Ohms	—	Characteristic impedance of free space

Constants are given in base units (*g*, *m*, etc.) rather than the natural units for the unit system (*kg*, *cm*, etc.). For example, when using the CGS unit system, the speed of light is given as 300Mm/s (rather than 30Gcm/s).

As shown, these constants are partitioned into two *unit systems*: *mks* and *cgs*. Only those constants that are associated with the active unit system and those that are not associated with any unit system are available when creating a new quantity. You can activate a unit system using *set_unit_system()*. Doing so deactivates the previous system. By default, the *mks* system is active.

You can create your own constants and unit systems using *add_constant()*:

```
>>> from quantiphy import Quantity, add_constant
>>> add_constant(Quantity(": 211.061140539mm // wavelength of hydrogen line"))

>>> hy_wavelength = Quantity('')
```

(continues on next page)

(continued from previous page)

```
>>> print(hy_wavelength.render(show_label=True))
 = 211.06 mm - wavelength of hydrogen line
```

In this case is the name given in the quantity is used when creating the constant. You can also specify an alias as an argument to *add_constant*.

```
>>> add_constant(
...     Quantity(" = 211.061140539mm # wavelength of hydrogen line"),
...     alias='lambda h'
... )

>>> hy_wavelength = Quantity('lambda h')
>>> print(hy_wavelength.render(show_label=True))
 = 211.06 mm - wavelength of hydrogen line
```

It is not necessary to specify both the name and the alias, one is sufficient; the constant is accessible using either. Notice that the alias does not actually become part of the constant, it is only used for looking up the constant.

By default, user defined constants are not associated with a unit system, meaning that they are always available regardless of which unit system is being used. However, when creating a constant you can specify one or more unit systems for the constant. You need not limit yourself to the predefined *mks* and *cgs* unit systems. You can specify multiple unit systems either by specifying a list of strings for the unit systems, or by specifying one string that would contain more than one name once split.

```
>>> from quantiphy import Quantity, add_constant, set_unit_system

>>> add_constant(Quantity(4.80320427e-10, 'Fr'), 'q', 'esu gaussian')
>>> add_constant(Quantity(1.602176487e-20, 'abC'), alias='q', unit_systems='emu')
>>> q_mks = Quantity('q')
>>> set_unit_system('cgs')
>>> q_cgs = Quantity('q')
>>> set_unit_system('esu')
>>> q_esu = Quantity('q')
>>> set_unit_system('gaussian')
>>> q_gaussian = Quantity('q')
>>> set_unit_system('emu')
>>> q_emu = Quantity('q')
>>> set_unit_system('mks')
>>> print(q_mks, q_cgs, q_esu, q_gaussian, q_emu, sep='\n')
160.22e-21 C
480.32 pFr
480.32 pFr
480.32 pFr
16.022e-21 abC
```

7.1.9 Preferences

QuantiPhy supports a wide variety of preferences that control its behavior. For example, when rendering quantities you can control the number of digits used (*prec*), whether SI scale factors are used (*form*), whether the units are included (*show_units*), etc. Similar preferences also control the conversion of strings into quantities, which can help disambiguate whether a suffix represents a scale factor or a unit. The list of available preferences and their descriptions are given in the description of the *Quantity.set_prefs()* method.

To set a preference, use the *Quantity.set_prefs()* class method. You can set more than one preference at once:

```
>>> Quantity.set_prefs(prec=6, map_sf={'u': ''})
```

This statements tells *QuantiPhy* to use 7 digits (the *prec* plus 1) and to output rather u for the 10^{-6} scale factor.

Setting preferences to *None* returns them to their default values:

```
>>> Quantity.set_prefs(prec=None, map_sf=None)
```

The preferences are changed on the class itself, meaning that they affect any instance of that class regardless of whether they were instantiated before or after the preferences were set. If you would like to have more than one set of preferences, then you should subclass *Quantity*. For example, imagine a situation where you have different types of quantities that would naturally want different preferences:

```
>>> class Temperature(Quantity):
...     units = 'C'
>>> Temperature.set_prefs(prec=1, known_units='K', spacer='')

>>> class Frequency(Quantity):
...     units = 'Hz'
>>> Frequency.set_prefs(prec=5, spacer='')

>>> frequencies = []
>>> for each in '-25.3 999987.7, 25.1  1000207.1, 74.9  1001782.3'.split(','):
...     temp, freq = each.split()
...     frequencies.append((Temperature(temp),  Frequency(freq)))

>>> for temp, freq in frequencies:
...     print(f'{temp:4}  {freq}')
-25C  999.988kHz
 25C  1.00021MHz
 75C  1.00178MHz
```

In this example, a subclass is created that is intended to report in concentrations.

```
>>> class Concentration(Quantity):
...     pass
>>> Concentration.set_prefs(
...     map_sf = dict(u=' PPM', n= ' PPB', p=' PPT'),
...     show_label = True,
... )

>>> pollutants = dict(CO=5, SO2=20, NO2=0.10)
>>> concentrations = [Concentration(v, scale=1e-6, name=k) for k, v in pollutants.
↪items()]
>>> for each in concentrations:
```

(continues on next page)

(continued from previous page)

```
...     print(each)
CO = 5 PPM
SO2 = 20 PPM
NO2 = 100 PPB
```

Alternately, you can simply set the preferences as attributes when creating the sublclasses. For example:

```
>>> class Dollars(Quantity):
...     units = '$'
...     prec = 2
...     form = 'fixed'
...     show_commas = True
...     minus = Quantity.minus_sign
...     strip_zeros = False
```

When a subclass is created, the preferences active in the main class are copied into the subclass. Subsequent changes to the preferences in the main class do not affect the subclass.

You can also go the other way and override the preferences on a specific quantity.

```
>>> print(hy_wavelength)
211.06 mm

>>> hy_wavelength.show_label = True
>>> print(hy_wavelength)
 = 211.06 mm - wavelength of hydrogen line
```

This is often the way to go with quantities that have logarithmic units such as decibels (dB) or shannons (Sh) (or the related bit, digits, nats, hartleys, etc.). In these cases use of SI scale factors is often undesired.

```
>>> gain = Quantity(0.25, 'dB')
>>> print(gain)
250 mdB

>>> gain.form = 'fixed'
>>> print(gain)
0.25 dB
```

To retrieve a preference, use the *Quantity.get_pref()* class method. This is useful with *known_units*. Normally setting *known_units* overrides the existing units. You can simply add more with:

```
>>> Quantity.set_prefs(known_units=Quantity.get_pref('known_units') + ['K'])
```

A variation on *Quantity.set_prefs()* is *Quantity.prefs()*. It is basically the same, except that it is meant to work with Python's *with* statement to temporarily override preferences:

```
>>> with Quantity.prefs(form='fixed', show_units=False, prec=2):
...     for time, temp in data:
...         print(f"{time:<7} {temp}")
0       505.37
600     477.59
1200    455.37
```

(continues on next page)

(continued from previous page)

```
>>> print(f"Final temperature = {data[-1][1]} @ {data[-1][0]}.")
Final temperature = 455.37 K @ 1.2 ks.
```

Notice that the specified preferences only affected the table, not the final printed values, which were rendered outside the *with* statement.

If you are using *QuantiPhy* in a large package with multiple modules and more than one includes *Quantity*, you may find that the preferences are not shared between the modules. This occurs because each module gets its own independent version of *Quantity*. To work around this issue you would create your own module that imports from *QuantiPhy*. Each of the packages' modules then import from your new module rather than directly from *QuantiPhy*. For example, consider creating a local module named *quantity.py*:

```
from quantiphy import *
import locale

# Base preferences
loc_conv = locale.localeconv()
radix = loc_conv['decimal_point']
comma = loc_conv['thousands_sep']
Quantity.set_prefs(radix=radix, comma=comma, known_units='K')

# Alternate preference sets
preferences = dict(
    user = dict(
        # assumes a user is reading values on a terminal with fixed-width font
        form = 'si',
        map_sf = Quantity.map_sf_to_greek,
        prec = 4,
        spacer = ' ',
        strip_radix = True,
        minus = Quantity.minus_sign,
        show_units = True,
    ),
    sphinx = dict(
        # assumes values are to be rendered with a variable-with font by Sphinx
        form = 'si',
        map_sf = Quantity.map_sf_to_sci_notation,
        prec = 4,
        spacer = Quantity.narrow_non_breaking_space,
        minus = Quantity.minus_sign,
        strip_radix = True,
        show_units = True,
    ),
    code_with_si = dict(
        # assumes values are to be rendered to code that accepts sf but not units
        form = 'sia',
        map_sf = None,
        prec = 'full',
        spacer = '',
        minus = '-'.minus_sign,
        strip_radix = 'cover',  # assures quantities are always treated as reals
    )
    code_without_si = dict(
```

(continues on next page)

(continued from previous page)

```
        # assumes values are to be rendered to code that does not accept sf or units
        form = 'eng',
        map_sf = None,
        prec = 'full',
        spacer = '',
        minus = '-'.minus_sign,
        strip_radix = 'cover',  # assures quantities are always treated as reals
    )
)

def set_quantity_defaults(choice):
    Quantity.set_prefs(**peferences[choice])

set_quantity_defaults('user')
```

Now, in the other modules, you would simply import from *quantity* rather than *quantiphy*:

```
from quantity import Quantity, QuantiPhyError, set_quantity_defaults
```

Then, if you change the preferences using *set_quantity_defaults* from one module, the preferences are changed for all modules.

7.1.10 Localization

Quantiphy provides 7 preferences that help with localization: *radix*, *comma*, *plus*, *minus*, *inf*, *nan*, and *spacer*.

radix
: The decimal point; generally `.` or `,`.

comma
: The thousands separator; generally `,`, `.`, `_` or a narrow non-breaking space.

plus
: *QuantitPhy* does not use plus signs when rendering quantities either on the mantissa or the exponent. But it will accept this symbol as a plus signs when converting strings to quantities.

minus
: The symbol used to indicate a negative number; generally `-` or `.` This symbol is also accepted as a minus signs when converting strings to quantities.

inf
: The symbol or word that signifies infinity; generally `inf` or ∞.

nan
: The symbol or word that indicates a NaN or Not-a-Number; generally `NaN` or `nan`.

spacer
: The character used to separate the mantissa from trailing units, or scale factor combined with units: generally `` `` or *Quantity.narrow_non_breaking_space*. *spacer* does not affect how strings are converted quantities, where the spacer is optional and may ether be a space, a non-breaking space, a thin space, or a narrow non-breaking space.

By default *QuantiPhy* uses `.`, `,`, `+`, `-`, `inf`, `nan` and `` `` as the defaults. These are all simple ASCII characters. They work as expected for the numbers normally used in programming, such as `-5.17e+06`.

Both *radix* and *comma* affect the way stings are converted to quantities and they way quantities are rendered. When interpreting a string as a number, *QuantiPhy* first strips the *comma* character from the string and then replaces the *radix* character with `.`.

If you prefer to use `,` for your radix, you generally have two choices. With the first, *radix* is set to `,` and *comma* to `.`. This allows you to properly read and write numbers like €100.000.000,00 but misinterpretes a number if it uses `.` as the radix.

```
>>> Quantity.set_prefs(radix=',', comma='.')
>>> q1 = Quantity('€100.000,00')
>>> q2 = Quantity('€100000.00')
>>> print(q1, q2, sep='\n')
€100k
€10M
```

With the second, *radix* is set to `,` and *comma* to ''. This allows both `,` and `.` to be used as the radix, so €100,000 and €100.000 have the same value. However, it fails for numbers that use `.` as the thousands separator.

```
>>> Quantity.set_prefs(radix=',', comma='')
>>> q1 = Quantity('€100,000')
>>> q2 = Quantity('€100.000')
>>> print(q1, q2, sep='\n')
€100
€100
```

You can automatically adapt to local conventions using the Python *locale* package:

```
>>> from quantiphy import Quantity
>>> import locale

>>> loc_conv = locale.localeconv()
>>> radix = loc_conv['decimal_point']
>>> comma = loc_conv['thousands_sep']
>>> Quantity.set_prefs(radix=radix, comma=comma)

>>> q = Quantity('€100.000')
>>> print(q)
€100

>>> print(f"radix is '{radix}'\ncomma is '{comma}'")
radix is '.'
comma is ''
```

You can convert from one convention to the other by changing *radix* and *comma* on the fly:

```
>>> with Quantity.prefs(radix=',', comma='.'):
...     q = Quantity('€100.000.000,00')
>>> with Quantity.prefs(radix='.', comma=','):
...     print(f'{q:#,.2p}')
€100,000,000.00
```

7.1.11 Formatting Tabular Data

When creating tables it is often desirable to align the decimal points of the numbers, and perhaps align the units. You can use the *number_fmt* to arrange this. *number_fmt* is a format string that if specified is used to convert the components of a number into the final number. You can control the widths and alignments of the components to implement specific arrangements. *number_fmt* is passed to the string *format* function with named arguments: *whole*, *frac* and *units*, which contains the integer part of the number, the fractional part including the decimal point, and the units including the scale factor. More information about the content of the components can be found in *Quantity.set_prefs()*.

For example, you can align the decimal point and units of a column of numbers like this:

```
>>> lengths = [
...     Quantity(l)
...     for l in '1mm, 10mm, 100mm, 1.234mm, 12.34mm, 123.4mm'.split(',')
... ]

>>> with Quantity.prefs(number_fmt='{whole:>3}{frac:<4} {units}'):
...     for l in lengths:
...         print(l)
  1     mm
 10     mm
100     mm
  1.234 mm
 12.34  mm
123.4   mm
```

You can also give a function as the value for *number_fmt* rather than a string. It would be called with *whole*, *frac* and *units* as arguments given in that order. The function is expected to return the assembled number as a string. For example:

```
>>> def fmt_num(whole, frac, units):
...     return '{mantissa:<5} {units}'.format(mantissa=whole+frac, units=units)

>>> with Quantity.prefs(number_fmt=fmt_num):
...     for l in lengths:
...         print(l)
1     mm
10    mm
100   mm
1.234 mm
12.34 mm
123.4 mm
```

If there are multiple columns it might be necessary to apply a different format to each column. In this case, it often makes sense to create a subclass of Quantity for each column that requires distinct formatting:

```
>>> def format_temperature(whole, frac, units):
...     return '{:>5} {:<5}'.format(whole+frac, units)

>>> class Temperature(Quantity):
...     units = 'C'
>>> Temperature.set_prefs(
...     prec = 1, known_units = 'K', number_fmt = format_temperature
... )
```

(continues on next page)

(continued from previous page)

```
>>> class Frequency(Quantity):
...     units = 'Hz'
>>> Frequency.set_prefs(prec=5, number_fmt = '{whole:>3}{frac:<6} {units}')

>>> frequencies = []
>>> for each in '-25.3 999987.7, 25.1 1000207.1, 74.9 1001782.3'.split(','):
...     temp, freq = each.split()
...     frequencies.append((Temperature(temp),  Frequency(freq)))

>>> for temp, freq in frequencies:
...     print(temp, freq)
  -25 C     999.988   kHz
   25 C       1.00021 MHz
   75 C       1.00178 MHz
```

7.1.12 Extract Quantities

It is possible to put a collection of quantities in a text string and then use the *Quantity.extract()* method to parse the quantities and return them in a dictionary. For example:

```
>>> design_parameters = '''
...     Fref (f) = 156 MHz  - Reference frequency
...     Kdet = 88.3 uA       - Gain of phase detector
...     Kvco = 9.07 GHz/V    - Gain of VCO
... '''
>>> quantities = Quantity.extract(design_parameters)

>>> Quantity.set_prefs(
...     label_fmt = '{n} = {v}',
...     label_fmt_full = '{V:<18}  # {d}',
...     show_label = 'f',
... )
>>> for k, q in quantities.items():
...     print(f'{k}: {q}')
Fref: f = 156 MHz        # Reference frequency
Kdet: Kdet = 88.3 uA      # Gain of phase detector
Kvco: Kvco = 9.07 GHz/V   # Gain of VCO
```

The string is processed one line at a time and may contain any number of quantity definitions. Blank lines are ignored. Each non-blank line is passed through *assign_rec* to determine if it is recognized as an assignment. If it is recognized, the *assign_rec* named fields (*name*, *qname*, *val*, and *desc*) are used when creating the quantity. The default recognizer allows you to separate the name from the value with either '=' or ':'. It allows you to separate the value from the description using '—', '–', '//', or '#'. These substrings are also used to introduce comments, so you could start a line with '#' and it would be treated as a comment. If the line is not recognized, then it is ignored.

In this example, the first line is nonconforming and so is ignored. The second *Kvdo* line is a comment, the comment character and anything beyond is ignored. Finally, empty lines are ignored.

```
>>> design_parameters = '''
...     PLL Design Parameters
```

(continues on next page)

(continued from previous page)

```
...
...     Fref = 156 MHz      - Reference frequency
...     Kdet = 88.3 uA      - Gain of phase detector
...     Kvco = 9.07 GHz/V   - Gain of VCO
...     // Kvco = 5 GHz/V     - Gain of VCO
...     N = 128             - Divide ratio
...     Fout = N*Fref "Hz"  - Output Frequency
... '''
>>> globals().update(Quantity.extract(design_parameters))

>>> print(f'{Fref:S}\n{Kdet:S}\n{Kvco:S}\n{N:S}\n{Fout:}')
Fref = 156 MHz      # Reference frequency
Kdet = 88.3 uA      # Gain of phase detector
Kvco = 9.07 GHz/V   # Gain of VCO
N = 128             # Divide ratio
Fout = 19.968 GHz   # Output Frequency
```

In this case the output of the *Quantity.extract()* call is fed into globals().update() so as to add the quantities into the module namespace, making the quantities accessible as local variables. This is an example of how simulation scripts could be written. The system and simulation parameters would be gathered together at the top into a multiline string, which would then be read and loaded into the local name space. It allows you to quickly give a complete description of a collection of parameters when the goal is to put something together quickly in an expressive manner. Another example of this ideas is shown a bit further down where the module docstring is used to contain the quantity definitions.

Here is an example that uses this feature to read parameters from a file. This is basically the same idea as above, except the design parameters are kept in a separate file. It also subclasses *Quantity* to create a version that displays the name and description by default.

```
>>> from quantiphy import Quantity, InvalidNumber
>>> from inform import os_error, fatal, display

>>> class VerboseQuantity(Quantity):
...     show_label = 'f'
...     label_fmt = '{n} = {v}'
...     label_fmt_full = '{V:<18} - {d}'

>>> filename = '.parameters'
>>> try:
...     with open(filename) as f:
...         globals().update(VerboseQuantity.extract(f.read()))
... except OSError as e:
...     fatal(os_error(e))
... except InvalidNumber as e:
...     fatal(e, culprit=filename)

>>> print(Fref, Kdet, Kvco, N, Fout, sep='\n')
Fref = 156 MHz      - Reference frequency
Kdet = 88.3 uA      - Gain of phase detector (Imax)
Kvco = 9.07 GHz/V   - Gain of VCO
N = 128             - Divide ratio
Fout = 19.968 GHz   - Output Frequency
```

With *Quantity.extract()* the values of quantities can be given as a expression that contains previously defined

quantities (or *physical constants* or select mathematical constants (pi, tau, , or). You can follow an expression with a string to give the units. Finally, you can use the *predefined* argument to pass in a dictionary of named values that can be used in your expressions. For example:

```
#!/usr/bin/env python3
>>> __doc__ = """
... Simulates a second-order  modulator with the following parameter values:
...
...     Fclk = Fxtal/4 "Hz"                 - clock frequency
...     Fin = 200kHz                        - input frequency
...     Vin = 950mV                         - input voltage amplitude (peak)
...     gain1 = 0.5V/V                      - gain of first integrator
...     gain2 = 0.5V/V                      - gain of second integrator
...     Vmax = 1V                           - quantizer maximum input voltage
...     Vmin = -1V                          - quantizer minimum input voltage
...     levels = 5                          - quantizer output levels
...     Tstop = 2/Fin "s"                   - simulation stop time
...     Tstart = -1/Fin 's'                 - initial transient interval (discarded)
...     file_name = 'out.wave'              - output filename
...     sim_name = f'{Fclk:q}  Modulator'  - simulation name
...
... The values given above are used in the simulation; no further
... modification of the code given below is required when changing
... these parameters.
... """

>>> from quantiphy import Quantity

>>> Fxtal = Quantity('200 MHz')
>>> parameters = Quantity.extract(__doc__, predefined=dict(Fxtal=Fxtal))
>>> globals().update(parameters)

>>> with Quantity.prefs(
...     label_fmt = '{n} = {v}',
...     label_fmt_full = '{V:<18} - {d}',
...     show_label = 'f',
... ):
...     print('Simulation parameters:')
...     for k, v in parameters.items():
...         try:
...             print(f'    {v:Q}')
...         except ValueError:
...             print(f'    {k} = {v!s}')
Simulation parameters:
    Fclk = 50 MHz      - clock frequency
    Fin = 200 kHz      - input frequency
    Vin = 950 mV       - input voltage amplitude (peak)
    gain1 = 500 mV/V   - gain of first integrator
    gain2 = 500 mV/V   - gain of second integrator
    Vmax = 1 V         - quantizer maximum input voltage
    Vmin = -1 V        - quantizer minimum input voltage
    levels = 5         - quantizer output levels
    Tstop = 10 us      - simulation stop time
```

(continues on next page)

(continued from previous page)

```
    Tstart = -5 us     - initial transient interval (discarded)
    file_name = out.wave
    sim_name = 50 MHz  Modulator
```

Notice in this case the parameters were specified and read out of the docstring at the top of the file. In this way, the parameters become very easy to set and the documentation is always up to date. Ignore the fact that the docstring is assigned to *__doc__*. That was a hack that was needed to make the example executable from within the documentation.

7.1.13 Translating Quantities

Quantity.all_from_conv_fmt() recognizes conventionally formatted numbers and quantities embedded in text and reformats them using *Quantity.render()*. This is an difficult task in general, and so some constraints are placed on the values to make them easier to distinguish. Specifically, the units, if given, must be simple and immediately adjacent to the number. Units are simple if they only consist of letters and underscores. The characters °, Å, and are also allowed. So '47e3Ohms', '50_Ohms' and '1.0e+12' are recognized as quantities, but '50 Ohms' and '12m/s' are not.

Besides the text to be translated, `all_from_conv_fmt()` takes the same arguments as *render()*, though they must be given as named arguments.

```
>>> test_results = '''
... Applying stimulus @ 2.00500000e-04s: V(in) = 5.000000e-01V.
... Pass @ 3.00500000e-04s: V(out): expected=2.00000000e+00V, measured=1.99999965e+00V,␣
↪diff=3.46117130e-07V.
... '''.strip()

>>> Quantity.set_prefs(spacer='')
>>> translated = Quantity.all_from_conv_fmt(test_results)
>>> print(translated)
Applying stimulus @ 200.5us: V(in) = 500mV.
Pass @ 300.5us: V(out): expected=2V, measured=2V, diff=346.12nV.
```

Quantity.all_from_si_fmt() is similar, except that it recognizes quantities formatted with either a scale factor or units and ignores plain numbers. Again, units are expected to be simple and adjacent to their number.

```
>>> Quantity.set_prefs(spacer='')
>>> translated_back = Quantity.all_from_si_fmt(translated, form='eng')
>>> print(translated_back)
Applying stimulus @ 200.5e-6s: V(in) = 500e-3V.
Pass @ 300.5e-6s: V(out): expected=2V, measured=2V, diff=346.12e-9V.
```

Notice in the translations the quantities lost resolution. This is avoided if you use 'full' precision:

```
>>> translated = Quantity.all_from_conv_fmt(test_results, prec='full')
>>> print(translated)
Applying stimulus @ 200.5us: V(in) = 500mV.
Pass @ 300.5us: V(out): expected=2V, measured=1.99999965V, diff=346.11713nV.
```

7.1.14 Equivalence

You can determine whether a value is equivalent to that of a quantity using *Quantity.is_close()*. The value may be given as a quantity, a real number, or a string. The two values need not be identical, they just need to be close to be deemed equivalent. The *reltol* and *abstol* preferences are used to determine if they are close.

```
>>> h_line.is_close(h_line)
True

>>> h_line.is_close(h_line + 1)
True

>>> h_line.is_close(h_line + 1e4)
False
```

Quantity.is_close() returns true if the units match and if:

abs(a - b) <= max(reltol * max(abs(a), abs(b)), abstol)

where a and b represent *other* and the numeric value of the underlying quantity.

By default, *is_close()* looks at the both the value and the units if the argument has units. In this way if you compare two quantities with different units, the *is_close()* test will always fail if their units differ. This behavior can be overridden by specifying *check_units*.

```
>>> Quantity('$10').is_close('10 USD')
False

>>> Quantity('$10').is_close('10 USD', check_units=False)
True
```

7.1.15 Negligible Values

QuantiPhy can round small values to zero when rendering them, which can help to reduce visual clutter. You can specify the size of a negligible value as a preference using *Quantity.set_prefs()* or *Quantity.prefs()*, or you can specify it locally using *Quantity.render()*. Any quantity whose absolute value is smaller than the specified value is rendered as zero with the underlying value remaining unchanged.

```
>>> from quantiphy import Quantity
>>> from math import exp

>>> Vt = 0.025852
>>> def cond(v):
...     return Quantity(1e-27 * exp(v/Vt)/Vt, '')

>>> Quantity.set_prefs(prec=2)
>>> for i in range(11):
...     v = Quantity(i/5, 'V')
...     print(f'{v:>6}: {cond(v):>10}, {v:>26}: {cond(v).render(negligible=1e-3):>10}')
   0 V: 38.7e-27 ,                          0 V:          0
200 mV: 88.6e-24 ,                     200 mV:          0
```

(continues on next page)

(continued from previous page)

```
400 mV:  203e-21 ,                  400 mV:        0
600 mV:     465 a,                  600 mV:        0
800 mV:    1.06 p,                  800 mV:        0
   1 V:    2.44 n,                     1 V:        0
 1.2 V:    5.58 u,                   1.2 V:        0
 1.4 V:    12.8 m,                   1.4 V:    12.8 m
 1.6 V:     29.3 ,                   1.6 V:     29.3
 1.8 V:      67 k,                   1.8 V:      67 k
   2 V:     153 M,                     2 V:     153 M
```

7.1.16 Exceptional Values

QuantiPhy supports NaN (not a number) and infinite values:

```
>>> inf = Quantity('inf Hz')
>>> print(inf)
inf Hz

>>> nan = Quantity('NaN Hz')
>>> print(nan)
NaN Hz
```

You can test whether the value of the quantity is infinite or is not-a-number using *Quantity.is_infinite()* or *Quantity.is_nan()*. These method return a rendered value for the number without units if they are true and None otherwise:

```
>>> h_line.is_infinite()

>>> inf.is_infinite()
'inf'

>>> h_line.is_nan()

>>> nan.is_nan()
'NaN'
```

The rendered value is affected by the *inf* and *nan* preferences or attributes:

```
>>> inf.inf = '∞'
>>> inf.is_infinite()
'∞'
```

7.1.17 Exceptions

The way exceptions are defined in *QuantiPhy* has changed. Initially, the standard Python exceptions were used to indicate errors. For example, a *ValueError* was raised by *Quantity* if it were passed a string it cannot convert into a number. Now, a variety of *QuantiPhy* specific exceptions are used to indicate specific errors. However, these exceptions subclass the corresponding Python error for compatibility with existing code. It is recommended that new code catch the *QuantiPhy* specific exceptions rather than the generic Python exceptions as their use will be deprecated in the future.

QuantiPhy employs the following exceptions:

***ExpectedQuantity*:**
Subclass of *QuantiPhyError* and *ValueError*. Used by *add_constant()*.

Raised when the value is either not an instance of *Quantity* or a string that can be converted to a quantity.

***IncompatiblePreferences*:**
Subclass of *QuantiPhyError* and *ValueError*. Used by *Quantity* constructor.

Raised when comma and radix preference is the same.

***IncompatibleUnits*:**
Subclass of *QuantiPhyError* and *TypeError*. Used by *Quantity.add()*.

Raised when the units of contribution do not match those of underlying quantity.

***InvalidNumber*:**
Subclass of *QuantiPhyError*, *ValueError*, and *TypeError*. Used by *Quantity()*.

Raised if the value given could not be converted to a number.

***InvalidRecognizer*:**
Subclass of *QuantiPhyError* and *KeyError*. Used by *Quantity()*.

The *assign_rec* preference is expected to be a regular expression that defines one or more named fields, one of which must be *val*. This exception is raised when the current value of *assign_rec* does not satisfy this requirement.

***MissingName*:**
Subclass of *QuantiPhyError* and *NameError*. Used by *add_constant()*.

Raised when *alias* was not specified and no name was available from *value*.

***UnknownConversion*:**
Subclass of *QuantiPhyError* and *KeyError*.

Used by *UnitConversion.convert()*, *Quantity()*, *Quantity.scale()*, *Quantity.render()*, *Quantity.fixed()*, *Quantity.format()*, *Quantity.binary()*, *as_real()*, *as_tuple()*, *render()*, *fixed()*, and *binary()*.

Raised when a unit conversion was requested and there is no corresponding unit converter.

***UnknownFormatKey*:**
Subclass of *QuantiPhyError* and *KeyError*. Used by *Quantity.render()*, *Quantity.fixed()*, and *Quantity.format()*.

The *label_fmt* and *label_fmt_full* are expected to be format strings that may interpolate certain named arguments. The valid named arguments are *n* for name, *v* for value, and *d* for description. This exception is raised when some other name is used for an interpolated argument.

***UnknownPreference*:**
Subclass of *QuantiPhyError* and *KeyError*. Used by *Quantity.set_prefs()*, *Quantity.get_pref()*, and *Quantity.prefs()*.

Raised when the name given for a preference is unknown.

UnknownScaleFactor:
: Subclass of *QuantiPhyError* and *ValueError*. Used by *Quantity()*, *Quantity.set_prefs()*, or *Quantity.prefs()*.

  The *input_sf* preference gives the list of scale factors that should be accepted. This exception is raised if *input_sf* contains an unknown scale factor.

UnknownUnitSystem:
: Subclass of *QuantiPhyError* and *KeyError*. Used by *set_unit_system()*.

  Raised when the name given does not correspond to a known unit system.

QuantiPhy defines a common base exception, *QuantiPhyError*, that all specific exceptions derive from. This allows you to simplify your exception handling if you are not interested in distinguishing between the specific errors:

```
>>> from quantiphy import Quantity, QuantiPhyError

>>> try:
...     q = Quantity('tweed')
... except QuantiPhyError as e:
...     print(str(e))
'tweed': not a valid number.
```

The alternative would be to catch each error individually:

```
>>> from quantiphy import (
...     Quantity, InvalidNumber, UnknownScaleFactor,
...     UnknownConversion, InvalidRecognizer
... )

>>> try:
...     q = Quantity('tweed')
... except (InvalidNumber, UnknownScaleFactor, UnknownConversion, InvalidRecognizer) as␣
↪e:
...     print(str(e))
'tweed': not a valid number.
```

QuantiPhy provides uniform access methods for its exceptions. You can access all the unnamed arguments passed to the exception using the *args* attribute, you can access the named arguments using *kwargs*, and you can create a customized message that incorporates the arguments using *QuantiPhyError.render()* method. The argument to *render* is a format string that can access both the unnamed and named arguments:

```
>>> try:
...     q = Quantity('tweed')
... except InvalidNumber as e:
...     print(e.render('{}: no es un número valido.'))
... except UnknownScaleFactor as e:
...     print(e.render('factor de escala desconocido.'))
... except UnknownConversion as e:
...     if 'to_units' in e.kwargs:
...         if 'from_units' in e.kwargs:
...             template = 'incapaz de convertir entre {} y {}'
...         else:
...             template = 'incapaz de convertir a {}'
...     else:
...         template = 'incapaz de convertir de {}'
```

(continues on next page)

(continued from previous page)

```
...     print(e.render(template))
... except InvalidRecognizer as e:
...     print(e.render("el reconocedor no contiene la clave 'val'"))
tweed: no es un número valido.
```

Alternately, if you wish to globally replace the default *QuantiPhy* error messages, you can simply override the `_template` attribute for the exceptions:

```
>>> InvalidNumber._template = '{!r}: no es un número valido.'
>>> UnknownScaleFactor._template = 'factor de escala desconocido.'
>>> UnknownConversion._template = (
...     'incapaz de convertir entre ‘{to_units}’ y ‘{from_units}’',
...     'incapaz de convertir a ‘{to_units}’',
...     'incapaz de convertir de ‘{from_units}’',
... )
>>> InvalidRecognizer._template = "el reconocedor no contiene la clave ‘val’"

>>> try:
...     q = Quantity('tweed')
... except QuantiPhyError as e:
...     print(e.render())
'tweed': no es un número valido.
```

As shown in *UnknownConversion*, `_template` may be given as a tuple of format strings, in which case the first one for which all arguments are available is used.

7.2 Classes and Functions

7.2.1 Quantities

class quantiphy.**Quantity**(*value*, *model=None*, ***, *units=None*, *scale=None*, *binary=None*, *name=None*, *desc=None*, *ignore_sf=None*, *params=None*)

Create a physical quantity.

A quantity is a number paired with a unit of measure.

Parameters

- **value** (*real, string or quantity*) – The value of the quantity. If a string, it may be the name of a pre-defined constant or it may be a number that may be specified with SI scale factors and/or units. For example, the following are all valid: ‘2.5ns’, ‘1.7 MHz’, ‘1e6’, ‘2.8_V’, ‘1e4 m/s’, ‘$10_000’, ‘42’, ‘’, etc. The string may also have name and description if they are provided in a way recognizable by *assign_rec*. For example, ‘trise: 10ns — rise time’ or ‘trise = 10ns # rise time’ would work with the default recognizer.
- **model** (*quantity or string*) – Used to pick up any missing attibutes (*units*, *name*, *desc*). May be a quantity or a string. If model is a quantity, only its units would be taken. If model is a string, it is split. Then, if there is one item, it is taken to be *units*. If there are two, they are taken to be *name* and *units*. And if there are three or more, the first two are taken to the be *name* and *units*, and the remainder is taken to be *description*.
- **units** (*str*) – Overrides the units taken from *value* or *model*.
- **scale** (*float, tuple, func, string, or quantity*) –

  - If a float or quantity, it multiplies by the given value to compute the value of the quantity. If a quantity, the units are ignored.
  - If a tuple, the first value, a float, is treated as a scale factor and the second value, a string, is take to be the units of the quantity.
  - If a function, it takes two arguments, the given value and the units and it returns two values, the value and units of the quantity.
  - If a string, it is taken to the be desired units. This value along with the units of the given value are used to select a known unit conversion, which is applied to create the quantity.
- **name** (*str*) – Overrides the name taken from *value* or *model*.
- **desc** (*str*) – Overrides the desc taken from *value* or *model*.
- **ignore_sf** (*bool*) – Assume the value given within a string does not employ a scale factors. In this way, '1m' is interpreted as 1 meter rather than 1 milli.
- **binary** (*bool*) – Allow use of binary scale factors (Ki, Mi, Gi, Ti, Pi, Ei, Zi, Yi).
- **params** – Parameters to be used in scaling. May be scalar, tuple, or dictionary.

Raises

- ***UnknownConversion*(*QuantiPhyError*, KeyError)** – A unit conversion was requested and there is no corresponding unit converter.
- ***InvalidRecognizer*(*QuantiPhyError*, KeyError)** – Assignment recognizer (*assign_rec*) does not match at least the value (*val*).
- ***UnknownScaleFactor*(*QuantiPhyError*, ValueError)** – Unknown scale factor or factors.
- ***InvalidNumber*(*QuantiPhyError*, ValueError, TypeError)** – Not a valid number.
- ***IncompatiblePreferences*(*QuantiPhyError*, ValueError)** – *radix* and *comma* must differ.

You can use *Quantity* to create quantities from floats, strings, or other You can use *Quantity* to create quantities from floats, strings, or other quantities. If a float is given, *model* or *units* would be used to specify the units.

Examples:

```
>>> from quantiphy import Quantity
>>> from math import pi, tau
>>> newline = '''
... '''

>>> fhy = Quantity('1420.405751786 MHz')
>>> sagan = Quantity(pi*fhy, 'Hz')
>>> sagan2 = Quantity(tau*fhy, fhy)
>>> print(fhy, sagan, sagan2, sep=newline)
1.4204 GHz
4.4623 GHz
8.9247 GHz
```

You can use *scale* to scale the number or convert to different units when creating the quantity.

Examples:

```
>>> Tfreeze = Quantity('273.15 K', ignore_sf=True, scale='°C')
>>> print(Tfreeze)
0 °C

>>> Tboil = Quantity('212 °F', scale='°C')
>>> print(Tboil)
100 °C
```

Public Methods:

reset_prefs()	Reset preferences
set_prefs(**kwargs)	Set class preferences.
get_pref(name)	Get class preference.
prefs(**kwargs)	Set class preferences.
is_infinite()	Test value to determine if quantity is infinite.
is_nan()	Test value to determine if quantity is not a number.
as_tuple()	Return a tuple that contains the value as a float along with its units.
scale(scale[, cls])	Scale a quantity to create a new quantity.
add(addend[, check_units])	Create a new quantity that is the sum of the original and a contribution.
render(*[, form, show_units, prec, ...])	Convert quantity to a string.
fixed(*[, show_units, prec, show_label, ...])	Convert quantity to fixed-point string.
binary(*[, show_units, prec, show_label, ...])	Convert quantity to string using binary scale factors.
is_close(other[, reltol, abstol, check_units])	Are values equivalent?
format([template])	Convert quantity to string under the guidance of a template.
extract(text[, predefined])	Extract quantities.
map_sf_to_sci_notation(sf)	Render scale factors in scientific notation.
map_sf_to_greek(sf)	Render scale factors in Greek alphabet if appropriate.
all_from_conv_fmt(text[, only_e_notation])	Convert all numbers and quantities from conventional notation.
all_from_si_fmt(text, **kwargs)	Convert all numbers and quantities from SI notation.

---

add(*addend*, *check_units=False*)

Create a new quantity that is the sum of the original and a contribution.

Parameters

- **addend** (*real, quantity, string*) – The amount to add to the quantity.
- **check_units** (*boolean or 'strict'*) – If True, raise an exception if the units of the *addend* are not compatible with the underlying quantity. If the *addend* does not have units, then it is considered compatible unless *check_units* is 'strict'.

Raises

***IncompatibleUnits*(*QuantiPhyError*, TypeError)** – Units of contribution do not match those of underlying quantity.

Example:

```
>>> total = Quantity(0, '$')
>>> for contribution in [1.23, 4.56, 7.89]:
...     total = total.add(contribution)
>>> print(total)
$13.68
```

classmethod all_from_conv_fmt(*text*, *only_e_notation=False*, ***kwargs*)

Convert all numbers and quantities from conventional notation.

Only supports a subset of the conventional formats that *QuantiPhy* normally accepts. For example, leading units (ex. $1M) and embedded commas are not supported, and the radix is always '.'.

There may be a space between the number an units, but it cannot be a normal space. Only non-breaking, thin-non-breaking and thin spaces are allowed.

Parameters

- **text** (*str*) – A search and replace is performed on this text. The search looks for numbers and quantities in floating point or e-notation. They are replaced with the same number rendered as a quantity. To be recognized any units must be simple (only letters or underscores, no digits or symbols) and the units must be immediately adjacent to the number.
- **only_e_notation** (*bool*) – If true, only numbers that explicitly have exponents are converted (1e6Hz is converted, but not 1.6 or 2009). If False, numbers with or without exponents are converted (1e6Hz, 1.6 and 2009 are all converted.
- ****kwargs** – By default the numbers are rendered using the currently active preferences, but any valid argument to *Quantity.render()* can be passed in to control the rendering.

Returns
A copy of *text* where all numbers that were formatted conventionally have been reformatted.

Return type
str

Example:

```
>>> text = 'Applying stimulus @ 2.05000e-05s: V(in) = 5.00000e-01V.'
>>> with Quantity.prefs(spacer=''):
...     xlated = Quantity.all_from_conv_fmt(text)
...     print(xlated)
Applying stimulus @ 20.5us: V(in) = 500mV.
```

classmethod all_from_si_fmt(*text*, ***kwargs*)

Convert all numbers and quantities from SI notation.

Only supports a subset of the SI formats that *QuantiPhy* normally accepts. For example, leading units (ex. $1M) and embedded commas are not supported, and the radix is always '.'.

Parameters

- **text** (*str*) – A search and replace is performed on this text. The search looks for numbers and quantities formatted in SI notation (must have either a scale factor or units or both). They are replaced with the same number rendered as a quantity. To be recognized any units must be simple (only letters or underscores, no digits or symbols) and the units must be immediately adjacent to the number.
- ****kwargs** – By default the numbers are rendered using the currently active preferences, but any valid argument to *Quantity.render()* can be passed in to control the rendering.

Returns
A copy of *text* where all numbers that were formatted with SI scale factors have been reformatted.

Return type
str

Example:

```
>>> print(Quantity.all_from_si_fmt(xlated))
Applying stimulus @ 20.5 us: V(in) = 500 mV.

>>> print(Quantity.all_from_si_fmt(xlated, form='eng'))
Applying stimulus @ 20.5e-6 s: V(in) = 500e-3 V.
```

as_tuple()

Return a tuple that contains the value as a float along with its units.

Example:

```
>>> period = Quantity('10ns')
>>> period.as_tuple()
(1e-08, 's')
```

binary(*, *show_units=None*, *prec=None*, *show_label=None*, *strip_zeros=None*, *strip_radix=None*, *scale=None*)

Convert quantity to string using binary scale factors.

When in range the number is divided by some integer power of 1024 and the appropriate scale factor is added to the quotient, where the scale factors are '' for 0 powers of 1024, 'Ki' for 1, 'Mi' for 2, 'Gi' for 3, 'Ti' for 4, 'Pi' for 5, 'Ei' for 6, 'Zi' for 7 and 'Yi for 8. Outside this range, the number is converted to a string using a simple floating point format.

Within the range the number of significant figures used is equal to prec+1. Outside the range, prec give the number of figures to the right of the decimal point.

Parameters

- **show_units** (*bool*) – Whether the units should be included in the string.
- **prec** (*integer or 'full'*) – The desired precision (number of digits to the right of the radix when normalized). If specified as 'full', *full_prec* is used as the number of digits (and not the originally specified precision as with render).
- **show_label** (*'f', 'a', or boolean*) – Add the name and possibly the description when rendering a quantity to a string. Either *label_fmt* or *label_fmt_full* is used to label the quantity.
  - neither is used if *show_label* is False,
  - otherwise *label_fmt* is used if quantity does not have a description or if *show_label* is 'a' (short for abbreviated),
  - otherwise *label_fmt_full* is used if *show_desc* is True or *show_label* is 'f' (short for full).
- **strip_zeros** (*boolean*) – Remove contiguous zeros from end of fractional part. If not specified, the global *strip_zeros* setting is used.
- **strip_radix** (*boolean*) – Remove radix if there is nothing to the right of it. If not specified, the global *strip_radix* setting is used.

- **scale** (*real, pair, function, or string*) –
  - If a float, it scales the displayed value (the quantity is multiplied by scale before being converted to the string).
  - If a tuple, the first value, a float, is treated as a scale factor and the second value, a string, is take to be the units of the displayed value.
  - If a function, it takes two arguments, the value and the units of the quantity and it returns two values, the value and units of the displayed value.
  - If a string, it is taken to the be desired units. This value along with the units of the quantity are used to select a known unit conversion, which is applied to create the displayed value.

Raises

- ***UnknownConversion*(*QuantiPhyError*, KeyError)** – A unit conversion was requested and there is no corresponding unit converter.
- ***UnknownFormatKey*(*QuantiPhyError*, KeyError)** – 'label_fmt' or 'label_fmt_full' contains an unknown format key.

Example:

```
>>> t = Quantity('mem = 16 GiB - amount of physical memory', binary=True)
>>> print(
...     t.binary(),
...     t.binary(prec=3, strip_zeros=False),
...     t.binary(show_label=True, scale='b'), sep=newline)
16 GiB
16.00 GiB
mem = 128 Gib
```

classmethod extract(*text*, *predefined=None*, ***kwargs*)

Extract quantities.

Takes a string that contains quantity definitions, one per line, and returns those quantities in a dictionary.

Parameters

- **text** (*str*) – The string that contains the quantities, one definition per line. Each is parsed by *assign_rec*. By default, the lines are assumed to be of the form:

```
[<name> [(<qname>)] = <value>] [- <description>]
```

where '=' may be replaced by ':' and '—' (the em-dash) may be replaced by '–', '//' or '#'. In addition, brackets delimit optional fields and parentheses represent literal parentheses. Each of the fields are allowed be largely arbitrary strings.

The brackets indicate that the name/value pair and the description is optional. However, <name> must be given if <value> is given.

<name>:
the name is used as a key for the value.

<qname>:
the name taken by the quantity.

<value>:
A number with optional units (ex: 3 or 1pF or 1 k); the units need not be a simple identifier (ex: 9.07 GHz/V).

The value may also be an expression. When giving an expression, you may follow it with a string surrounded by double quotes, which is taken as the units. For example: Tstop = 5/Fin "s". The expressions may only contain value defined previously in the same set of definitions, values contained in *predefined*, physical constants, the mathematical constants pi and tau (2*pi), which may be named or , or number literals without scale factors or units. The units should not include a scale factor.

When processing the value, it is passed as an argument to Quantity, if cannot be converted to a quantity, then it is treated as a Python expression.

<description>:
Optional textual description (ex: Frequency of hydrogen line).

Blank lines and any line that does not contain a value are ignored. So with the default *assign_rec*, lines with the following form are ignored:

```
- comment
-- comment
# comment
// comment
```

- **predefined** (*dict*) – A dictionary of predefined values. When specified, these values become available to be used in the expressions that give values to the values being defined. You can use *locals()* as this argument to make all local variables available.

  You can specify both values and functions. For example, `predefined=dict(sqrt=sqrt)` allows `sqrt` to be used in expressions.

- ****kwargs** – Any argument that can be passed to Quantity can be passed to this function, and are in turn passed to Quantity as the quantities are created. This can be used, for example, to allow the binary scale factors.

Returns
a dictionary of quantities for the values specified in the argument.

Return type
dict

Example:

```
>>> sagan_frequencies = r'''
...     - Carl Sagan's SETI frequencies of high interest
...
...     f_hy = 1420.405751786 MHz - Hydrogen line frequency
...     f_sagan1 = *f_hy "Hz" - Sagan's first frequency
...     f_sagan2 = *f_hy "Hz" - Sagan's second frequency
... '''
>>> freqs = Quantity.extract(sagan_frequencies)
>>> for f in freqs.values():
...     print(f.render(show_label='f'))
f_hy = 1.4204 GHz - Hydrogen line frequency
f_sagan1 = 4.4623 GHz - Sagan's first frequency
f_sagan2 = 8.9247 GHz - Sagan's second frequency

>>> globals().update(freqs)
>>> print(f_hy, f_sagan1, f_sagan2, sep=newline)
1.4204 GHz
```

(continues on next page)

(continued from previous page)

```
4.4623 GHz
8.9247 GHz
```

fixed(*, *show_units=None*, *prec=None*, *show_label=None*, *show_commas=None*, *strip_zeros=None*, *strip_radix=None*, *scale=None*)

Convert quantity to fixed-point string.

Parameters

- **show_units** (*bool*) – Whether the units should be included in the string.
- **prec** (*integer or 'full'*) – The desired precision (one plus this value is the desired number of digits). If specified as 'full', the full original precision is used.
- **show_label** (*'f', 'a', or boolean*) – Add the name and possibly the description when rendering a quantity to a string. Either *label_fmt* or *label_fmt_full* is used to label the quantity.
  - neither is used if *show_label* is False,
  - otherwise *label_fmt* is used if quantity does not have a description or if *show_label* is 'a' (short for abbreviated),
  - otherwise *label_fmt_full* is used if *show_desc* is True or *show_label* is 'f' (short for full).
- **show_commas** – Add commas to whole part of mantissa, every three digits. If not specified, the global *strip_zeros* setting is used.
- **strip_zeros** (*boolean*) – Remove contiguous zeros from end of fractional part. If not specified, the global *strip_zeros* setting is used.
- **strip_radix** (*boolean*) – Remove radix if there is nothing to the right of it. If not specified, the global *strip_radix* setting is used.
- **scale** (*real, pair, function, or string*) –
  - If a float, it scales the displayed value (the quantity is multiplied by scale before being converted to the string).
  - If a tuple, the first value, a float, is treated as a scale factor and the second value, a string, is take to be the units of the displayed value.
  - If a function, it takes two arguments, the value and the units of the quantity and it returns two values, the value and units of the displayed value.
  - If a string, it is taken to the be desired units. This value along with the units of the quantity are used to select a known unit conversion, which is applied to create the displayed value.

Raises

- ***UnknownConversion*(*QuantiPhyError*, KeyError)** – A unit conversion was requested and there is no corresponding unit converter.
- ***UnknownFormatKey*(*QuantiPhyError*, KeyError)** – 'label_fmt' or 'label_fmt_full' contains an unknown format key.

Example:

```
>>> t = Quantity('Total = $1000000.00 - the total')
>>> print(
...     t.fixed(),
```

(continues on next page)

(continued from previous page)

```
...     t.fixed(show_commas=True),
...     t.fixed(show_units=False), sep=newline)
$1000000
$1,000,000
1000000

>>> print(
...     t.fixed(prec=2, strip_zeros=False, show_commas=True),
...     t.fixed(prec=6),
...     t.fixed(strip_zeros=False, prec=6), sep=newline)
$1,000,000.00
$1000000
$1000000.000000

>>> print(
...     t.fixed(strip_zeros=False, prec='full'),
...     t.fixed(show_label=True),
...     t.fixed(show_label='f'), sep=newline)
$1000000.00
Total = $1000000
Total = $1000000 - the total

>>> print(
...     t.fixed(scale=(1/10000, 'BTC')),
...     t.fixed(scale=(1/1000, 'ETH')),
...     t.fixed(scale=(1/1000, 'ETH'), show_units=False), sep=newline)
100 BTC
1000 ETH
1000
```

format(*template=''*)

Convert quantity to string under the guidance of a template.

Supports the normal floating point and string format types as well some new ones. If the format code is given in upper case, *label_fmt* is used to add the name and perhaps description to the result.

Parameters

template (*str*) – the format string.

Raises

- ***UnknownFormatKey*(*QuantiPhyError*, KeyError)** – 'label_fmt' or 'label_fmt_full' contains an unknown format key.
- ***UnknownConversion*(*QuantiPhyError*, KeyError)** – A unit conversion was requested and there is no corresponding unit converter.

The format is specified using A#W,.PTU where:

```
A   is a character and gives the alignment: either '', '>', '<', or '^'
#   is a literal hash that if present indicates that
    trailing zeros and radix should not be suppressed from fractional part.
W   is an integer and gives the width of the final string
,   is a literal comma, it indicates that the whole part of the
    mantissa should be partitioned into groups of three digits
```

(continues on next page)

(continued from previous page)

```
    separated by commas
.P  is a literal period followed by an integer that gives the precision
T   is a character and gives the type: choose from p, q, r, s, e, f, g, u, n, d,
↪ ...
U   is a string that must match a known unit, it invokes scaling
```

Each of these component pieces is optional.

If:

```
q = Quantity('f = 1420.405751786 MHz - hydrogen line')
```

then:

```
q: quantity [si=y, units=y, label=n] (ex: 1.4204 GHz)
Q: quantity [si=y, units=y, label=y] (ex: f = 1.4204 GHz)
r: real [si=y, units=n, label=n] (ex: 1.4204G)
R: real [si=y, units=n, label=y] (ex: f = 1.4204G)
 : [label=n] (ex: 1.4204 GHz)
p: fixed-point [fixed=y, units=y, label=n] (ex: 1420405751.7860 Hz)
P: fixed-point [fixed=y, units=y, label=y] (ex: f = 1420405751.7860 Hz)
s: string [label=n] (ex: 1.4204 GHz)
S: string [label=y] (ex: f = 1.4204 GHz)
e: exponential form [si=n, units=n, label=n] (ex: 1.4204e9)
E: exponential form [si=n, units=n, label=y] (ex: f = 1.4204e9)
f: float [label=n] (ex: 1420400000.0000)
F: float [label=y] (ex: f = 1420400000.0000)
g: generalized float [label=n] (ex: 1.4204e+09)
G: generalized float [label=y] (ex: f = 1.4204e+09)
u: units only (ex: Hz)
n: name only (ex: f)
d: description only (ex: hydrogen line)
```

classmethod get_pref(*name*)

Get class preference.

Returns the value of given preference.

Parameters

name (*str*) – Name of the desired preference. See *Quantity.set_prefs()* for list of preferences.

Raises

UnknownPreference(*QuantiPhyError*, **KeyError**) – unknown preference.

Example:

```
>>> Quantity.set_prefs(known_units='au')
>>> known_units = Quantity.get_pref('known_units')
>>> known_units.append('pc')
>>> Quantity.set_prefs(known_units=known_units)
>>> print(Quantity.get_pref('known_units'))
['au', 'pc']
```

is_close(*other*, *reltol=None*, *abstol=None*, *check_units=True*)

Are values equivalent?

Indicates whether the value of a quantity or real number is equivalent to that of a quantity. The two values need not be identical, they just need to be close to be deemed equivalent.

Parameters

- **other** (*quantity, real, or string*) – The value to compare against.
- **reltol** (*float*) – The relative tolerance. If not specified. the *reltol* preference is used, which defaults to 1u.
- **abstol** (*float*) – The absolute tolerance. If not specified. the *abstol* preference is used, which defaults to 1p.
- **check_units** (*bool*) – If True (the default), and if *other* is a quantity, compare the units of the two values, if they differ return False. Otherwise only compare the numeric values, ignoring the units.

Returns

Returns true if `abs(a - b) <= max(reltol * max(abs(a), abs(b)), abstol)` where `a` and `b` represent *other* and the numeric value of the underlying quantity.

Return type

bool

Example:

```
>>> print(
...     c.is_close(c),                      # should pass, is identical
...     c.is_close(c+1),                    # should pass, is close
...     c.is_close(c+1e4),                  # should fail, not close
...     c.is_close(Quantity(c+1, 'm/s')),   # should pass, is close
...     c.is_close(Quantity(c+1, 'Hz')),    # should fail, wrong units
...     c.is_close('299.7925 Mm/s'),        # should pass, is close
... )
True True False True False True
```

is_infinite()

Test value to determine if quantity is infinite. Returns a representation of the number (sign combined with self.inf) if value is infinite and None otherwise.

Example:

```
>>> inf = Quantity('inf Hz')
>>> inf.is_infinite()
'inf'
```

is_nan()

Test value to determine if quantity is not a number. Returns a representation of the number (sign combined with self.nan) if value is not a number and None otherwise.

Example:

```
>>> nan = Quantity('-nan Hz')
>>> nan.is_nan()
'NaN'
```

static map_sf_to_greek(*sf*)

Render scale factors in Greek alphabet if appropriate.

Pass this dictionary to *map_sf* preference if you prefer µ rather than u.

Example:

```
>>> with Quantity.prefs(map_sf=Quantity.map_sf_to_greek):
...     print(Quantity('mu0').render(show_label='f'))
μ₀ = 1.2566 μH/m - permeability of free space
```

static map_sf_to_sci_notation(*sf*)

Render scale factors in scientific notation.

Pass this function to *map_sf* preference if you prefer your large and small numbers in classic scientific notation. It also causes 'u' to be converted to 'µ'. Set *form* to 'eng' to format all numbers in scientific notation.

Example:

```
>>> with Quantity.prefs(map_sf=Quantity.map_sf_to_sci_notation, show_label='f'):
...     print(
...         Quantity('k').render(),
...         Quantity('mu0').render(),
...         Quantity('mu0').render(form='eng'),
...         sep=newline,
...     )
k = 13.806×10^-24 J/K - Boltzmann's constant
μ₀ = 1.2566 μH/m - permeability of free space
μ₀ = 1.2566×10^-6 H/m - permeability of free space
```

classmethod prefs(***kwargs*)

Set class preferences.

This is just like *Quantity.set_prefs()*, except it is designed to work as a context manager, meaning that it is meant to be used with Python's *with* statement. It allows preferences to be set to new values temporarily. They are reset upon exiting the *with* statement. For example:

```
>>> with Quantity.prefs(ignore_sf=True):
...     t = Quantity('600_000 K')
>>> t_bad = Quantity('600_000 K')
>>> print(t, t_bad, sep=newline)
600 kK
600M
```

See *Quantity.set_prefs()* for list of available arguments.

Raises

- ***UnknownPreference*(*QuantiPhyError*, KeyError)** – Unknown preference.
- ***UnknownScaleFactor*(*QuantiPhyError*, ValueError)** – Unknown scale factor or factors.

render(*, *form=None*, *show_units=None*, *prec=None*, *show_label=None*, *strip_zeros=None*, *strip_radix=None*, *scale=None*, *negligible=None*)

Convert quantity to a string.

Parameters

- **form** (*str*) – Specifies the form to use for representing numbers by default. Choose from 'si', 'sia', 'eng', 'fixed', and 'binary'. As an example 0.25 A is represented with 250 mA when form is 'si', as 250e-3 A when form is 'eng', and with 0.25 A when from is 'fixed'. 'sia' (SI ASCII) is like 'si', but causes *map_sf* preference to be ignored. 'binary' is like 'sia', but specifies that binary scale factors be used. Default is 'si'.
- **show_units** (*bool*) – Whether the units should be included in the string.
- **prec** (*integer or 'full'*) – The desired precision (one plus this value is the desired number of digits). If specified as 'full', the full original precision is used.
- **show_label** (*'f', 'a', or boolean*) – Add the name and possibly the description when rendering a quantity to a string. Either *label_fmt* or *label_fmt_full* is used to label the quantity.
  - neither is used if *show_label* is False,
  - otherwise *label_fmt* is used if quantity does not have a description or if *show_label* is 'a' (short for abbreviated),
  - otherwise *label_fmt_full* is used if *show_desc* is True or *show_label* is 'f' (short for full).
- **strip_zeros** (*boolean*) – Remove contiguous zeros from end of fractional part. If not specified, the global *strip_zeros* setting is used.
- **strip_radix** (*boolean*) – Remove radix if there is nothing to the right of it. If not specified, the global *strip_radix* setting is used.
- **scale** (*real or dict*) –
  - If a float or a quantity, it scales the displayed value (the quantity is multiplied by scale before being converted to the string). If a quantity, the units are ignored.
  - If a tuple, the first value, a float, is treated as a scale factor and the second value, a string, is take to be the units of the displayed value.
  - If a function, it takes two arguments, the value and the units of the quantity and it returns two values, the value and units of the displayed value.
  - If a string, it is taken to the be desired units. This value along with the units of the quantity are used to select a known unit conversion, which is applied to create the displayed value.
- **negligible** – If the absolute value of the quantity is equal to or smaller than *negligible*, it is rendered as 0. To make *negligible* a function of the units of the quantity, pass a dictionary where the keys are the units and the values are the value to use for negligible. A key of '' is used for quantities with no units and a key of None provides a default value for *negligible* that is used if the units of the quantity are not found in the dictionary.

Raises

- ***UnknownConversion*(*QuantiPhyError*, KeyError)** – A unit conversion was requested and there is no corresponding unit converter.
- ***UnknownFormatKey*(*QuantiPhyError*, KeyError)** – 'label_fmt' or 'label_fmt_full' contains an unknown format key.

Example:

```
>>> c = Quantity('c')
>>> print(
...     c.render(),
...     c.render(form='si'),
```

(continues on next page)

(continued from previous page)

```
...     c.render(form='eng'),
...     c.render(form='fixed'),
...     c.render(show_units=False),
...     c.render(prec=6),
...     c.render(prec='full'),
...     c.render(show_label=True),
...     c.render(show_label='f'),
...     sep=newline
... )
299.79 Mm/s
299.79 Mm/s
299.79e6 m/s
299792458 m/s
299.79M
299.7925 Mm/s
299.792458 Mm/s
c = 299.79 Mm/s
c = 299.79 Mm/s - speed of light

>>> print(
...     Tfreeze.render(scale='°F'),
...     Tboil.render(scale='°F'),
...     sep=newline
... )
32 °F
212 °F
```

classmethod reset_prefs()

Reset preferences

Resets all preferences to the current preferences of the parent class. If there is no parent class, they are reset to their defaults.

scale(*scale*, *cls=None*)

Scale a quantity to create a new quantity.

Parameters

- **scale** (*real, pair, function, string, or quantity*) –
  - If a float, it scales the existing value (a new quantity is returned whose value equals the existing quantity multiplied by scale. In this case the scale is assumed unitless and so the units of the new quantity are the same as those of the existing quantity).
  - If a tuple, the first value, a float, is treated as a scale factor and the second value, a string, is taken to be the units of the new quantity.
  - If a function, it takes two arguments, the value to be scaled and its units. The value is guaranteed to be a Quantity that includes the units, so the second argument is redundant and will eventually be deprecated. The function returns two values, the value and units of the new value.
  - If a string, it is taken to the be desired units, perhaps with a scale factor. This value along with the units of the quantity are used to select a known unit conversion, which is applied to create the new value.

  - If a quantity, the units are ignored and the scale is treated as if were specified as a unitless float.
  - If a subclass of *Quantity* that includes units, the units are taken to the be desired units and the behavior is the same as if a string were given, except that *cls* defaults to the given subclass.

- **cls** (*class*) – Class to use for return value. If not given, the class of self is used it the units do not change, in which case *Quantity* is used.

Raises

***UnknownConversion*(*QuantiPhyError*, KeyError)** – A unit conversion was requested and there is no corresponding unit converter.

Example:

```
>>> Tf = Tfreeze.scale('°F')
>>> Tb = Tboil.scale('°F')
>>> print(Tf, Tb, sep=newline)
32 °F
212 °F
```

classmethod set_prefs(***kwargs*)

Set class preferences.

Any values not passed in are left alone. Pass in *None* to reset a preference to its default value.

Parameters

- **abstol** (*float*) – Absolute tolerance, used by *Quantity.is_close()* when determining equivalence. Default is 10^{12}.
- **accept_binary** (*bool*) – Allow use of binary scale factors (Ki, Mi, Gi, Ti, Pi, Ei, Zi, Yi). Default is False.
- **assign_rec** (*str*) – Regular expression used to recognize an assignment. Used in constructor and extract(). By default an '=' or ':' separates the name from the value and a '—', '–', '#', or '//' separates the value from the description, if a description is given. So the default recognizes the following forms:

```
'vel = 60 m/s'
'vel = 60 m/s - velocity'
'vel = 60 m/s -- velocity'
'vel = 60 m/s # velocity'
'vel = 60 m/s // velocity'
'vel: 60 m/s'
'vel: 60 m/s - velocity'
'vel: 60 m/s -- velocity'
'vel: 60 m/s # velocity'
'vel: 60 m/s // velocity'
```

The name, value, and description are identified in the regular expression using named groups the names *name*, *val* and *desc*. For example:

```
assign_req = r'(?P<name>.*+) = (?P<val>.*?) - (?P<desc>.*?)',
```

The regular expression is interpreted using the re.VERBOSE flag.

When used with *Quantity.extract()* there are a few more features.

First, you may also introduce comments using '—', '–', '#', or '//':

```
'- comment'
'-- comment'
'# comment'
'// comment'
```

Second, you can specify an alternate name using by placing in within parentheses following the name:

```
'wavelength () = 21 cm - wavelength of hydrogen line'
```

In this case, the name attribute for the quantity will be '' and the quantity will be filed in the output dictionary using 'wavelength' as the key. If the alternate name is not given, then 'wavelength' is used for the quantity name and dictionary key.

Third, the value may be an expression involving the previously specified values. When doing so, you can specify the units by following the value expression with a double-quoted string. The expressions may contain numeric literals, previously defined quantities, and the constants pi and tau. For example:

```
parameters = Quantity.extract(r'''
    Fin = 250MHz - frequency of input stimulus
    Tstop = 10/Fin "s" - simulation stop time
''')
```

In this example, the value for *Tstop* is given as an expression involving *Fin*.

- **comma** (*str*) – The character to be used as the thousands separator. It is very common to use a comma, but using a space, period, or an underscore can be used. For your convenience, you can access a non-breaking space using `Quantity.non_breaking_space`, `Quantity.narrow_non_breaking_space`, or `Quantity.thin_space`.
- **form** (*str*) – Specifies the form to use for representing numbers by default. Choose from 'si', 'sia', 'eng', 'fixed', and 'binary'. As an example, 0.25 A is represented with 250 mA when form is 'si', as 250e-3 A when form is 'eng', and with 0.25 A when from is 'fixed'. 'sia' (SI ASCII) is like 'si', but causes *map_sf* to be ignored. 'binary' is like 'sia', but specifies that binary scale factors be used. Default is 'si'.
- **full_prec** (*int*) – Default full precision in digits where 0 corresponds to 1 digit. Must be nonnegative. This precision is used when the full precision is requested and the precision is not otherwise known. Default is 12.
- **ignore_sf** (*bool*) – Whether all scale factors should be ignored by default when recognizing numbers. Default is False.
- **inf** (*str*) – The text to be used to represent infinity. By default its value is 'inf', but is often set to '∞' (the unicode infinity symbol). You can access the Unicode infinity symbol using `Quantity.infinity_symbol`.
- **input_sf** (*str*) – Which scale factors to recognize when reading numbers. The default is 'YZEPTGMKk_cmuμnpfazy'. You can use this to ignore the scale factors you never expect to reduce the chance of a scale factor/unit ambiguity. For example, if you expect to encounter temperatures in Kelvin and can do without 'K' as a scale factor, you might use 'TGMK_munpfa'. This also gets rid of the unusual scale factors.
- **keep_components** (*bool*) – Indicate whether components should be kept if quantity value was given as string. Doing so takes a bit of space, but allows the original precision of the number to be recreated when full precision is requested. Default is True.

- **known_units** (*list or string*) – List of units that are expected to be used in preference to a scale factor when the leading character could be mistaken as a scale factor. If a string is given, it is split at white space to form the list. When set, any previous known units are overridden. Default is empty.

- **label_fmt** (*str*) – Format string used when label is requested if the quantity does not have a description or if the description was not requested (if *show_desc* is False). Is passed through string .format() method. Format string takes two possible arguments named *n* and *v* for the name and value. A typical values include:

```
'{n} = {v}'    (default)
'{n}: {v}'
```

- **label_fmt_full** (*str*) – Format string used when label is requested if the quantity has a description and the description was requested (if *show_desc* is True). Is passed through string .format() method. Format string takes four possible arguments named *n*, *v*, *d* and *V* for the name, value, description, and value as formatted by *label_fmt*. Typical value include:

```
'{n} = {v} - {d}'    (default)
'{n} = {v} -- {d}'
'{n} = {v} # {d}'
'{n} = {v} // {d}'
'{n}: {v} - {d}'
'{n}: {v} -- {d}'
'{V} - {d}'
'{V} -- {d}'
'{V:<20}  # {d}'
```

  The last example shows the *V* argument with alignment and width modifiers. In this case the modifiers apply to the name and value after being they are combined with the *label_fmt*. This is typically done when printing several quantities, one per line, because it allows you to line up the descriptions.

- **map_sf** (*dictionary or function*) – Use this to change the way individual scale factors are rendered, ex: map_sf={'u': ''} to render micro using mu. If a function is given, it takes a single string argument, the nominal scale factor (which would be the exponent if no scale factor fits), and returns either a string or a tuple. The string is the desired scale factor. The tuple consists of the string and a flag. If the flag is True the string is treated as an exponent, otherwise it is treated as a scale factors. The difference between an exponent and a scale factor is that the spacer goes after an exponent and before a scale factor. *QuantiPhy* provides two predefined functions intended for use with *maps_sf*: *Quantity.map_sf_to_greek()* and *Quantity.map_sf_to_sci_notation()*. Default is empty.

- **minus** (*str*) – The text to be used as the minus sign. By default its value is '-', but is sometimes '' (the unicode minus sign). You can access the Unicode minus sign using `Quantity.minus_sign`.

  This preference only affects how numbers are rendered. Both - and the unicode are always accepted as a minus sign when interpreting strings as numbers.

- **nan** (*str*) – The text to be used to represent a value that is not-a-number. By default its value is 'NaN'.

- **negligible** (*real or dictionary*) – If the absolute value of the quantity is equal to or smaller than *negligible*, it is rendered as 0. To make *negligible* a function of the units of the quantity, pass a dictionary where the keys are the units and the values are the value

to use for negligible. A key of '' is used for quantities with no units and a key of None provides a default value for *negligible* that is used if the units of the quantity are not found in the dictionary.

- **number_fmt** (*dictionary or function*) – Format string used to convert the components of the number into the number itself. Normally this is not necessary. However, it can be used to perform special formatting that is helpful when aligning numbers in tables. It allows you to specify the widths and alignments of the individual components. There are three named components: *whole*, *frac*, and *units*. *whole* contains the portion of the mantissa to the left of the radix (decimal point). It is the whole mantissa if there is no radix. It also includes the sign and the leading units (currency symbols), if any. *frac* contains the radix and the fractional part. It also contains the exponent if the number has one. *units* contains the scale factor and units. The following value can be used to align both the radix and the units, and give the number a fixed width:

```
number_fmt = '{whole:>3s}{frac:<4s} {units:<3s}'
```

  The various widths and alignments could be adjusted to fit a variety of needs.

  It is also possible to specify a function as *number_fmt*, in which case it is passed the three values in order (*whole*, *frac* and *units*) and is expected to return the number as a string.

- **output_sf** (*str*) – Which scale factors to output, generally one would only use familiar scale factors. The default is 'TGMkmunpfa', which gets rid or the very large ('QRYZEP') and very small ('zyrq') scale factors that many people do not recognize. You can set this to *Quantity.all_sf* to configure *Quantity* to use all available output scale factors.

- **radix** (*str*) – The character to be used as the radix. By default it is '.'.

- **plus** (*str*) – The text to be used as the plus sign. By default it is '+', but is sometimes '' (the unicode full width plus sign) or '' to simply eliminate plus signs from numbers. You can access the Unicode full width plus sign using `Quantity.plus_sign`.

  This preference only affects how numbers are rendered. Both + and the unicode are always accepted as a plus sign when interpreting strings as numbers.

  QuantiPhy currently does not add leading plus signs to either mantissa or exponent, so this setting is ignored.

- **prec** (*int or str*) – Default precision in digits where 0 corresponds to 1 digit. Must be nonnegative. This precision is used when the full precision is not required. Default is 4.

- **preferred_units** (*dict*) – A dictionary that is used when looking up the preferred units when rendering. For example, if *preferred_units* contains the entry: {"": "Ohms Ohm ohms ohm"}, then when rendering a quantity with units "Ohms", "Ohm", "ohms", or "ohm", the units are rendered as "".

- **reltol** (*float*) – Relative tolerance, used by *Quantity.is_close()* when determining equivalence. Default is 10^6.

- **show_commas** (*bool*) – When rendering to fixed-point string, add commas to the whole part of the mantissa, every three digits. By default this is False.

- **show_desc** (*bool*) – Whether the description should be shown if it is available when showing the label. By default *show_desc* is False.

  Deprecated since version 2.1: Use `show_label='f'` instead.

- **show_label** (*'f', 'a', or bool*) – Add the name and possibly the description when rendering a quantity to a string. Either *label_fmt* or *label_fmt_full* is used to label the quantity.

  – Neither is used if *show_label* is False,

  - otherwise *label_fmt* is used if quantity does not have a description or if *show_label* is 'a' (short for abbreviated),
  - otherwise *label_fmt_full* is used if *show_desc* is True or *show_label* is 'f' (short for full).

- **spacer** (*str*) – The spacer text to be inserted in a string between the numeric value and the scale factor when units are present. Is generally specified to be '' or ' '; use the latter if you prefer a space between the number and the units. Generally using ' ' makes numbers easier to read, particularly with complex units, and using '' is easier to parse. You could also use a Unicode non-breaking space ' '. For your convenience, you can access a non-breaking space using `Quantity.non_breaking_space`, `Quantity.narrow_non_breaking_space`, or `Quantity.thin_space`.

  Certain units, as defined using the *tight_units* preference, cause the spacer to be suppressed.

- **strip_radix** (*bool or str*) – When rendering, strip the radix (decimal point) if not needed from numbers even if they could then be mistaken for integers.

  There are three valid values: *True*, *False*, and "cover". If *True*, the radix is removed if it is the last character in the mantissa, so 1 is rendered as "1". If *False*, it is not removed, so 1 is rendered as "1.". If "cover", the radix is replaced by ".0", so 1 is rendered as "1.0". Thus, "cover" is a variant of *False*; it also retains the radix but adds a 0 to avoid a 'hanging' radix.

  If this setting is False, the radix is still stripped if the number has a scale factor. The default value is True.

  Set *strip_radix* to False when generating output that will be read by a parser that distinguishes between integers and reals based on the presence of a decimal point or scale factor.

  Be aware that use of "cover" can give the impression of more precision than is intended. For example, 1.4 if rendered with *prec=0* would be "1.0", which suggests a precision of 1 rather than 0. This true only if *prec* is less than 3.

- **strip_zeros** (*bool*) – When rendering, strip off any unneeded zeros from the number. By default this is True.

  Set strip_zeros to False when you would like to indicated the precision of your numbers based on the number of digits shown.

- **tight_units** (*list of strings*) – The spacer is suppressed with these units. By default, this is done for: % ° ' " . Some add °F and °C as well.

- **unity_sf** (*str*) – The output scale factor for unity, generally '' or '_'. The default is '', but use '_' if you want there to be no ambiguity between units and scale factors. For example, 0.3 would be rendered as '300m', and 300 m would be rendered as '300_m'.

Raises

- ***UnknownPreference*(*QuantiPhyError*, KeyError)** – Unknown preference.
- ***UnknownScaleFactor*(*QuantiPhyError*, ValueError)** – Unknown scale factor or factors.

Example:

```
>>> mu0 = Quantity('mu0')
>>> print(mu0)
1.2566 uH/m

>>> Quantity.set_prefs(prec=6, map_sf={'u': ''})
>>> print(mu0)
```

(continues on next page)

(continued from previous page)

```
1.256637 H/m

>>> Quantity.set_prefs(prec=None, map_sf=None)
>>> print(mu0)
1.2566 uH/m
```

Quantity Functions

These functions are provided for those that prefer use *QuantiPhy* to convert numbers in strings directly to floats, rather than keep the values around as *Quantity* objects.

quantiphy.**as_real**(**args*, ***kwargs*)

Convert to real.

Takes the same arguments as *Quantity*, but returns a float rather than a Quantity. Takes one additional optional keyword argument ...

Parameters

cls (*class*) – Quantity subclass used to do the conversion. If not given, *Quantity* is used.

Examples:

```
>>> from quantiphy import as_real
>>> print(as_real('1 uL'))
1e-06

>>> print(as_real('1.2 mg/L', scale='M', params=74.55))
1.6096579476861166e-05
```

quantiphy.**as_tuple**(**args*, ***kwargs*)

Convert to tuple (value, units).

Takes the same arguments as *Quantity*, but returns a tuple consisting of the value and units. Takes one additional optional keyword argument ...

Parameters

cls (*class*) – Quantity subclass used to do the conversion. If not given, *Quantity* is used.

Examples:

```
>>> from quantiphy import as_tuple
>>> print(as_tuple('1 uL'))
(1e-06, 'L')

>>> print(as_tuple('1.2 mg/L', scale='M', params=74.55))
(1.6096579476861166e-05, 'M')
```

quantiphy.**render**(*value*, *units*, *params=None*, **args*, ***kwargs*)

Render value and units to string (SI scale factors format).

The first two arguments are the value and the units and are required. The remaining arguments are the same as those of *Quantity.render()*.

Examples:

```
>>> from quantiphy import render
>>> print(render(1e-6, 'L'))
1 uL

>>> print(render(16.097e-6, 'M', scale='g/L', params=74.55))
1.2 mg/L
```

quantiphy.**fixed**(*value*, *units*, *params=None*, **args*, ***kwargs*)

Render value and units to string (fixed-point format).

The first two arguments are the value and the units and are required. The remaining arguments are the same as those of *Quantity.fixed()*.

Example:

```
>>> from quantiphy import fixed
>>> print(fixed(1e7, '$', show_commas=True, strip_zeros=False, prec=2))
$10,000,000.00
```

quantiphy.**binary**(*value*, *units*, *params=None*, **args*, ***kwargs*)

Render value and units to string (binary scale factors format)

The first two arguments are the value and the units and are required. The remaining arguments are the same as those of *Quantity.binary()*.

Example:

```
>>> from quantiphy import binary
>>> print(binary(2**32, 'B'))
4 GiB
```

7.2.2 Unit Conversion

class quantiphy.**UnitConversion**(*to_units*, *from_units*, *slope=1*, *intercept=0*)

Public Methods:

activate()	Re-activate a unit conversion.
convert([value, from_units, to_units])	Convert value to quantity with new units.
clear_all()	Remove all previously defined unit conversions.
fixture(converter_func)	A decorator fixture for unit conversion functions that can be used when creating parametrized unit conversions.

---

activate()

Re-activate a unit conversion.

Normally it is not necessary to call this method, however it can be used re-activate a previously created unit conversion that has since been overridden by a different unit conversion with the same to and from units.

classmethod clear_all()

Remove all previously defined unit conversions.

convert(*value=1*, *from_units=None*, *to_units=None*)

Convert value to quantity with new units.

A convenience method. Normally it is not needed because once created, a unit conversion becomes directly accessible to quantities and can be used both when creating or rendering the quantity.

Parameters

- **value** – The value to convert. May be a real number or a quantity. Alternately, may simply be a string, in which case it is taken to be the from_units. If the value is not given it is taken to be 1.
- **from_units** (*str*) – The units to convert from. If not given, the class's first from_units are used.
- **to_units** (*str*) – The units to convert to. If not given, the class's first to_units are used.

If the from_units were found among the class's from_units, and the to_units were found among the class's to_units, then a forward conversion is performed.

If the from_units were found among the class's to_units, and the to_units were found among the class's from_units, then a reverse conversion is performed.

Raises

***UnknownConversion*(*QuantiPhyError*, KeyError)** – The given units are not supported by the underlying class.

Example:

```
>>> print(str(m2pc))
m ← 3.0857e+16*pc

>>> m = m2pc.convert()
>>> print(str(m))
30.857e15 m

>>> pc = m2pc.convert(m)
>>> print(str(pc))
1 pc

>>> m = m2pc.convert(pc)
>>> print(str(m))
30.857e15 m

>>> m2pc.convert(30.857e15, 'm')
Quantity('1 pc')

>>> m2pc.convert(1000, 'pc')
Quantity('30.857e18 m')

>>> m2pc.convert('pc')
Quantity('30.857e15 m')
```

static fixture(*converter_func*)

A decorator fixture for unit conversion functions that can be used when creating parametrized unit conversions.

Creates an argument list for the decorated function based on the type of value given for the *params* argument to *Quantity*.

If *params* is a dictionary or mapping, its values are passed as named parameters.

If *params* is a tuple or list, its values are passed as positional arguments.

Otherwise, the value of *params* is passed as the second argument.

In all cases, the value being converted (an instance of *Quantity*) is passed as the first argument to the decorated converter function.

For example, when performing conversions between the molarity of a solution and its concentration in terms of g/L, the molecular weight of the compound used to make the solution is needed:

```
>>> from quantiphy import Quantity, UnitConversion

>>> @UnitConversion.fixture
... def from_molarity(M, mw):
...     return M * mw

>>> @UnitConversion.fixture
... def to_molarity(g_L, mw):
...     return g_L / mw

>>> conv = UnitConversion('g/L', 'M', from_molarity, to_molarity)

>>> KCl_M = Quantity('1.2 mg/L', scale='M', params=74.55)
>>> print(KCl_M)
16.097 uM
>>> print(f"{KCl_M:qg/L}")
1.2 mg/L

>>> NaCl_M = Quantity('5.0 mg/L', scale='M', params=58.44277)
>>> print(NaCl_M)
85.554 uM
>>> print(f"{NaCl_M:qg/L}")
5 mg/L
```

However, if you want to convert between mass and molarity where the mass is the amount of a compound needed to create a solution of a particular volume with a particular concentration, both the molecular weight and the volume are required parameters:

```
>>> @UnitConversion.fixture
... def to_molarity(mass, vol, mw):
...     moles = mass/mw
...     return moles/vol

>>> @UnitConversion.fixture
... def to_grams(molarity, vol, mw):
...     return molarity*vol*mw

>>> conv = UnitConversion('g', 'M', to_grams, to_molarity)

>>> KCl_M = Quantity('1.2 g', scale='M', params=dict(mw=74.55, vol=0.250))
>>> print(KCl_M)
```

(continues on next page)

(continued from previous page)

```
64.386 mM
>>> print(f"{KCl_M:pg}")
1.2 g

>>> NaCl_M = Quantity('5.0 g', scale='M', params=dict(mw=58.44277, vol=0.250))
>>> print(NaCl_M)
342.22 mM
>>> print(f"{NaCl_M:pg}")
5 g
```

7.2.3 Constants and Unit Systems

quantiphy.**add_constant**(*value*, *alias=None*, *unit_systems=None*)

Create a new constant.

Save a quantity in such a way that it can later be recalled by name when creating new quantities.

Parameters

- **value** (*quantity*) – The value of the constant. Must be a quantity or a string that can be directly converted to a quantity.
- **alias** (*str*) – An alias for the constant. Can be used to access the constant from as an alternative to the name given in the value, which itself is optional. If the value has a name, specifying this name is optional. If both are given, the constant is accessible using either name. *alias* may also be a list of aliases.
- **unit_systems** (*list or str*) – Name or names of the unit systems to which the constant should be added. If given as a string, string will be split at white space to create the list. If a constant is associated with a unit system, it is only available when that unit system is active. You need not limit yourself to the predefined 'mks' and 'cgs' unit systems. Giving a name creates the corresponding unit system if it does not already exist. If *unit_systems* is not given, the constant is not associated with a unit system, meaning that it is always available regardless of which unit system is active.

Raises

- ***ExpectedQuantity*(*QuantiPhyError*, ValueError)** – *value* must be an instance of *Quantity* or it must be a string that can be converted to a quantity.
- ***MissingName*(*QuantiPhyError*, NameError)** – *alias* was not specified and no name was available from *value*.

The constant is saved under *name* if given, and under the name contained within *value* if available. It is not necessary to supply both names, one is sufficient.

Example:

```
>>> from quantiphy import Quantity, add_constant
>>> add_constant('f_hy = 1420.405751786 MHz - Frequency of hydrogen line')
>>> print(Quantity('f_hy').render(show_label='f'))
f_hy = 1.4204 GHz - Frequency of hydrogen line
```

quantiphy.**set_unit_system**(*unit_system*)

Activates a unit system.

The default unit system is 'mks'. Calling this function changes the active unit system to the one with the specified name. Only constants associated with the active unit system or not associated with a unit system are available for use.

Parameters
unit_system (*str*) – Name of the desired unit system.

Raises
UnknownUnitSystem(*QuantiPhyError*, **KeyError**) – *unit_system* does not correspond to a known unit system.

Example:

```
>>> from quantiphy import Quantity, set_unit_system
>>> set_unit_system('cgs')
>>> print(Quantity('h').render(show_label='f'))
h = 6.6261e-27 erg-s - Plank's constant

>>> set_unit_system('mks')
>>> print(Quantity('h').render(show_label='f'))
h = 662.61e-36 J-s - Plank's constant
```

7.2.4 Exceptions

exception quantiphy.**QuantiPhyError**(**args*, ***kwargs*)

QuantiPhy base exception.

All of the specific QuantiPhy exceptions subclass this exception.

render(*template=None*)

Convert exception to a string under guidance of format string.

Parameters
template (*str*) – This string, along with the positional and keyword arguments of the exception are passed to the Python format() function and the result is returned. *template* may also be a list of strings. In this case the first string found that renders without error is used. If *template* is not given, the exception is rendered with the built-in template.

exception quantiphy.**ExpectedQuantity**(**args*, ***kwargs*)

The value is required to be a Quantity or a string that can be converted to a Quantity.

exception quantiphy.**IncompatiblePreferences**(**args*, ***kwargs*)

Two preferences are not compatible

exception quantiphy.**IncompatibleUnits**(**args*, ***kwargs*)

The units of the contribution do not match those of the underlying quantity.

exception quantiphy.**InvalidNumber**(**args*, ***kwargs*)

The value given could not be converted to a number.

exception quantiphy.**InvalidRecognizer**(**args*, ***kwargs*)

The *assign_rec* preference is expected to be a regular expression that defines one or more named fields, one of which must be *val*. This exception is raised when the current value of *assign_rec* does not satisfy this requirement.

exception quantiphy.**MissingName**(**args*, ***kwargs*)

alias was not specified and no name was available from *value*.

exception quantiphy.**UnknownConversion**(**args*, ***kwargs*)

The given units are not supported by the underlying class, or a unit conversion was requested and there is no corresponding unit converter.

exception quantiphy.**UnknownFormatKey**(**args*, ***kwargs*)

The *label_fmt* and *label_fmt_full* are expected to be format strings that may interpolate certain named arguments. The valid named arguments are *n* for name, *v* for value, and *d* for description. This exception is raised when some other name is used for an interpolated argument.

exception quantiphy.**UnknownPreference**(**args*, ***kwargs*)

The name given for a preference is unknown.

exception quantiphy.**UnknownScaleFactor**(**args*, ***kwargs*)

The *input_sf* preference gives the list of scale factors that should be accepted on a number. The *output_sf* preference gives the list of scale factors that should be used when rendering numbers. This exception is raised if *input_sf* or *output_sf* contains an unknown scale factor.

exception quantiphy.**UnknownUnitSystem**(**args*, ***kwargs*)

The name given does not correspond to a known unit system.

7.3 Examples

7.3.1 Motivating Example

QuantiPhy is a light-weight package that allows numbers to be combined with units into quantities. Quantities are very commonly encountered when working with real-world systems when numbers are involved. And when encountered, the numbers often use SI scale factors to make them easier to read and write. Surprisingly, most computer languages do not support numbers in this form. This is even more surprising when you realize that this form is a very well established international standard and has been for more than 50 years.

When working with quantities, one often has to choose between using a form that is easy for computers to read or one that is easy for humans to read. For example, consider this table of critical frequencies needed in jitter tolerance measurements in optical communication:

```
>>> table1 = """
...     SDH     | Rate          | f1      | f2       | f3      | f4
...     --------+---------------+---------+----------+---------+--------
...     STM-1   | 155.52 Mb/s   | 500 Hz  | 6.5 kHz  | 65 kHz  | 1.3 MHz
...     STM-4   | 622.08 Mb/s   | 1 kHz   | 25 kHz   | 250 kHz | 5 MHz
...     STM-16  | 2.48832 Gb/s  | 5 kHz   | 100 kHz  | 1 MHz   | 20 MHz
...     STM-64  | 9.95328 Gb/s  | 20 kHz  | 400 kHz  | 4 MHz   | 80 MHz
...     STM-256 | 39.81312 Gb/s | 80 kHz  | 1.92 MHz | 16 MHz  | 320 MHz
... """
```

This table was formatted to be easily read by humans. If it were formatted for computers, the numbers would be given without units and in exponential notation because they have dramatically different sizes. For example, it might look like this:

```
>>> table2 = """
...     SDH     | Rate (b/s)    | f1 (Hz) | f2 (Hz)  | f3 (Hz) | f4 (Hz)
...     --------+---------------+---------+----------+---------+--------
...     STM-1   | 1.5552e8      | 5e2     | 6.5e3    | 6.5e3   | 1.3e6
...     STM-4   | 6.2208e8      | 1e3     | 2.5e3    | 2.5e5   | 5e6
```

(continues on next page)

(continued from previous page)

```
...     STM-16  | 2.48832e9     | 5e3     | 1e5      | 1e6     | 2e7
...     STM-64  | 9.95328e9     | 2e4     | 4e5      | 4e6     | 8e7
...     STM-256 | 3.981312e10   | 8e4     | 1.92e6   | 1.6e7   | 3.2e8
... """
```

This contains the same information, but it is much harder for humans to read and interpret. Often the compromise of partially scaling the numbers can be used to make the table easier to interpret:

```
>>> table3 = """
...     SDH     | Rate (Mb/s)   | f1 (kHz)| f2 (kHz) | f3 (kHz)| f4 (MHz)
...     --------+---------------+---------+----------+---------+---------
...     STM-1   | 155.52        | 0.5     | 6.5      | 65      | 1.3
...     STM-4   | 622.08        | 1       | 2.5      | 250     | 5
...     STM-16  | 2488.32       | 5       | 100      | 1000    | 20
...     STM-64  | 9953.28       | 20      | 400      | 4000    | 80
...     STM-256 | 39813.12      | 80      | 1920     | 16000   | 320
... """
```

This looks cleaner, but it involves perhaps even more effort to interpret because the values are distant from their corresponding scaling and units, because the large and small values are oddly scaled (0.5 kHz is more naturally given as 500Hz and 39813 MHz is more naturally given as 39.8 GHz), and because each column may have a different scaling factor. While these might seem like minor inconveniences on this table, they can become quite annoying as tables become larger or more numerous. This problem exists with both tables and graphs. Fundamentally the issue is that your eyes are naturally drawn to the number, but the numbers are not complete. Your eyes need to hunt further and it is not obvious where to hunt. If not next to the number, the scaling and units for the numbers may be found in the column headings, the axes, the labels, the title, the caption, or in the body of the text. The sheer number of places to look can dramatically slow the interpretation of the data. This problem does not exist in the first table where each number is complete as it includes both its scaling and its units. The eye gets the full picture on the first glance.

This last version of the table represents a very common mistake people make when presenting data. They feel that adding units and scale factors to each number adds clutter and wastes space and so removes them from the data and places them somewhere else. Doing so results in a data that perhaps is visually cleaner but is harder for the reader to interpret. All these tables contain the same information, but in the second two tables the readability has been traded off in order to make the data easier to read into a computer because in most languages there is no easy way to read numbers that have either units or scale factors.

QuantiPhy makes it easy to read and generate numbers with units and scale factors so you do not have to choose between human and computer readability. For example, the above tables could be read with the following code (it must be tweaked somewhat to handle tables 2 and 3):

```
>>> from quantiphy import Quantity

>>> # parse the table
>>> sdh = []
>>> lines = table1.strip().split('\n')
>>> for line in lines[2:]:
...     fields = line.split('|')
...     name = fields[0].strip()
...     rate = Quantity(fields[1])
...     critical_freqs = [Quantity(f) for f in fields[2:]]
...     sdh.append((name, rate, critical_freqs))

>>> # print the table in a form suitable for humans
```

(continues on next page)

(continued from previous page)

```
>>> for name, rate, freqs in sdh:
...     print('{:8s}: {:12q} {:9q} {:9q} {:9q} {:9q}'.format(name, rate, *freqs))
STM-1   :  155.52 Mb/s    500 Hz   6.5 kHz    65 kHz   1.3 MHz
STM-4   :  622.08 Mb/s     1 kHz    25 kHz   250 kHz     5 MHz
STM-16  :  2.4883 Gb/s     5 kHz   100 kHz     1 MHz    20 MHz
STM-64  :  9.9533 Gb/s    20 kHz   400 kHz     4 MHz    80 MHz
STM-256 :  39.813 Gb/s    80 kHz  1.92 MHz    16 MHz   320 MHz

>>> # print the table in a form suitable for machines
>>> for name, rate, freqs in sdh:
...     print('{:8s}: {:12.4e} {:9.2e} {:9.2e} {:9.2e} {:9.2e}'.format(name, rate,␣
↪*freqs))
STM-1   :   1.5552e+08     5e+02   6.5e+03   6.5e+04   1.3e+06
STM-4   :   6.2208e+08     1e+03   2.5e+04   2.5e+05     5e+06
STM-16  :   2.4883e+09     5e+03     1e+05     1e+06     2e+07
STM-64  :   9.9533e+09     2e+04     4e+05     4e+06     8e+07
STM-256 :   3.9813e+10     8e+04  1.92e+06   1.6e+07   3.2e+08

>>> # print the table in a compromise form
>>> for name, rate, freqs in sdh:
...     print(
...         '{:8s}: {:12.2f} {:9.1f} {:9.1f} {:9.1f} {:9.1f}'.format(
...             name, rate.scale(1e-6), freqs[0].scale(1e-3),
...             freqs[1].scale(1e-3), freqs[2].scale(1e-3), freqs[3].scale(1e-6)
...         )
...     )
STM-1   :       155.52       0.5       6.5        65       1.3
STM-4   :       622.08         1        25       250         5
STM-16  :      2488.32         5       100      1000        20
STM-64  :      9953.28        20       400      4000        80
STM-256 :     39813.12        80      1920     16000       320
```

The code reads the data and then produces three outputs. The first output shows that quantities can be displayed in easily readable forms with their units (approximates table1). The second output shows that the values are easily accessible for computation (approximates table2). Finally, the third output represents a compromise between being human and machine readable (approximates table3).

Quantity is used to convert a number string, such as '155.52 Mb/s' into an internal representation that includes the value and the units: 155.52e6 and 'b/s'. The scaling factor is properly interpreted. Once a value is converted to a *Quantity*, it can be treated just like a normal *float*. The main difference occurs when it is time to convert it back to a string. When doing so, the scale factor and units are included by default.

7.3.2 DRAM Prices

Here is a table that was found on the Internet that gives the number of bits of dynamic RAM a dollar would purchase over time:

```
>>> bits_per_dollar = '''
...     1973 490
...     1978 2780
...     1983 16400
...     1988 91800
```

(continues on next page)

(continued from previous page)

```
...     1993 368000
...     1998 4900000
...     2003 26300000
...     2008 143000000
...     2013 833000000
...     2018 5000000000
... '''
```

It is pretty easy to read in the early years, but by the turn of the millennium you have to start counting the zeros by hand to understand the number. And are those bits or bytes? Reformatting with *QuantiPhy* makes it much more readable:

```
>>> for line in bits_per_dollar.strip().split('\n'):
...     year, bits = line.split()
...     bits = Quantity(bits, 'b')
...     print(f'{year}  {bits:11.2q}   {bits:11.2qB}')
1973        490 b        61.2 B
1978      2.78 kb         348 B
1983      16.4 kb       2.05 kB
1988      91.8 kb       11.5 kB
1993       368 kb        46 kB
1998       4.9 Mb       612 kB
2003      26.3 Mb       3.29 MB
2008       143 Mb       17.9 MB
2013       833 Mb        104 MB
2018         5 Gb        625 MB
```

Notice that *bits* was printed twice. The first time the formatting code included a width specification, but in the second the desired unit of measure was specified (*B*), which caused the underlying value to be converted from bits to bytes.

It is important to recognize that *QuantiPhy* is using decimal rather than binary scale factors. So 5 GB is 5 gigabyte and not 5 gibibyte. In other words 5 GB represents $5{\times}10^9$ B and not $5{\times}2^{30}$ B. This table can be reformulated to use the binary scale factors by changing the *q* format characters to *b*:

```
>>> for line in bits_per_dollar.strip().split('\n'):
...     year, bits = line.split()
...     bits = Quantity(bits, 'b')
...     print(f'{year}  {bits:11.2b}   {bits:11.2bB}')
1973        490 b        61.2 B
1978     2.71 Kib         348 B
1983       16 Kib         2 KiB
1988     89.6 Kib      11.2 KiB
1993      359 Kib      44.9 KiB
1998     4.67 Mib       598 KiB
2003     25.1 Mib      3.14 MiB
2008      136 Mib        17 MiB
2013      794 Mib      99.3 MiB
2018     4.66 Gib       596 MiB
```

7.3.3 Thermal Voltage Example

In this example, quantities are used to represent all of the values used to compute the thermal voltage: $Vt = kT/q$. It is not terribly useful, but does demonstrate several of the features of *QuantiPhy*.

```
>>> from quantiphy import Quantity
>>> with Quantity.prefs(
...     show_label = 'f',
...     label_fmt = '{n} = {v}',
...     label_fmt_full = '{V:<18}  # {d}',
... ):
...     T = Quantity(300, 'T K ambient temperature')
...     k = Quantity('k')
...     q = Quantity('q')
...     Vt = Quantity(k*T/q, f'Vt V thermal voltage at {T:q}')
...     print(T, k, q, Vt, sep='\n')
T = 300 K           # ambient temperature
k = 13.806e-24 J/K  # Boltzmann's constant
q = 160.22e-21 C    # elementary charge
Vt = 25.852 mV      # thermal voltage at 300 K
```

The first part of this example imports `Quantity` and sets the *show_label*, *label_fmt* and *label_fmt_full* preferences to display both the value and the description by default. *label_fmt* is used when the description is not present and *label_fmt_full* is used when it is present. In *label_fmt* the `{n}` is replaced by the *name* and `{v}` is replaced by the value (numeric value and units). In *label_fmt_full*, the `{V:<18}` is replaced by the expansion of *label_fmt*, left justified with a field width of 18, and the `{d}` is replaced by the description.

The second part defines four quantities. The first is given in a very specific way to avoid the ambiguity between units and scale factors. In this case, the temperature is given in Kelvin (K), and normally if the temperature were given as the string '300 K', the units would be confused for the scale factor. As mentioned in *Ambiguity of Scale Factors and Units* the 'K' would be treated as a scale factor unless you took explicit steps. In this case, this issue is circumvented by specifying the units in the *model* along with the name and description. The *model* is also used when creating *Vt* to specify the name, units, and description.

The last part simply prints the four values. The *show_label* preference is set so that names and descriptions are printed along with the values. In this case, since all the quantities have descriptions, *label_fmt_full* is used to format the output.

7.3.4 Casual Time Units

This example shows how one could allow users to enter time durations using a variety of casual units of time. *QuantiPhy* only pre-defines conversions for time units that are unambiguous and commonly used in scientific computation, so that leaves out units like months and years. However, in many situations the goal is simplicity rather than precision. In such a situation, it is convenient to support any units a user may reasonably expect to use. In a casual setting it would be very unusual to use SI scale factors, so there use will be prohibited to allow a greater range of units (ex. m for minutes).

This example assumes that a collection of time duration values are contained in a configuration file, in this example represented by *configuration*. Normally these values would be contained in a separate file that is opened and read, but for the sake of simplicity in the example, the 'contents' of the file is just given as a multiline string. The user can give the durations using any units they like, but internally they are all converted to seconds.

```
>>> from quantiphy import Quantity, UnitConversion
>>> _ = UnitConversion('s', 'sec second seconds')
>>> _ = UnitConversion('s', 'm min minute minutes', 60)
>>> _ = UnitConversion('s', 'h hr hour hours', 60*60)
```

(continues on next page)

(continued from previous page)

```
>>> _ = UnitConversion('s', 'd day days', 24*60*60)
>>> _ = UnitConversion('s', 'w week weeks', 7*24*60*60)
>>> _ = UnitConversion('s', 'M month months', 30*24*60*60)
>>> _ = UnitConversion('s', 'y year years', 365*24*60*60)
>>> Quantity.set_prefs(ignore_sf=True)

>>> configuration = '''
...     time_to_live = 3 months
...     time_limit = 1 day
...     time_out = 10m
... '''
>>> limits = Quantity.extract(configuration)

>>> for k, v in limits.items():
...     print(f'{k} = {v:ps}')
time_to_live = 7776000 s
time_limit = 86400 s
time_out = 600 s
```

Notice that the return values from *UnitConversion* are captured in a variable (_) in the code above. This is not necessary. It is done in this case to satisfy the testing framework that tests the code found in this documentation; normally the return value is discarded.

Another example of using *QuantiPhy* to implement casual time units is the *remind* script, which reminds you to do something after a specified amount of time has passed. You can find remind on GitHub.

7.3.5 Unicode Text Example

In this example *QuantiPhy* formats quantities to be embedded in text. To make the text as clean as possible, *QuantiPhy* is configured to use Unicode scale factors and the Unicode narrow non-breaking space as the spacer. The non-breaking space prevents units from being placed on a separate line from their number, making the quantity easier to read. The plus and minus signs are also replaced by their Unicode forms.

```
>>> from quantiphy import Quantity
>>> import textwrap

>>> Quantity.set_prefs(
...     map_sf = Quantity.map_sf_to_sci_notation,
...     spacer = Quantity.narrow_non_breaking_space,
...     plus = Quantity.plus_sign,
...     minus = Quantity.minus_sign
... )

>>> constants = [
...     Quantity('h'),
...     Quantity('hbar'),
...     Quantity('k'),
...     Quantity('q'),
...     Quantity('c'),
...     Quantity('0C'),
...     Quantity('eps0'),
...     Quantity('mu0'),
```

(continues on next page)

(continued from previous page)

```
...     Quantity('0', 'K', scale='°C', desc='Absolute zero'),
... ]

>>> # generate some sentences that contain quantities
>>> sentences = [f'{q.desc.capitalize()} is {q}.' for q in constants]

>>> # combine the sentences into a left justified paragraph
>>> print(textwrap.fill('  '.join(sentences)))
Plank's constant is 662.61×10^36J-s.  Reduced plank's constant is
105.46×10^36J-s.  Boltzmann's constant is 13.806×10^24J/K.
Elementary charge is 160.22×10^21C.  Speed of light is 299.79Mm/s.
Zero degrees celsius is 273.15K.  Permittivity of free space is
8.8542pF/m.  Permeability of free space is 1.2566µH/m.  Absolute
zero is 273.15°C.
```

When rendered in your browser with a variable width font, the result looks like this:

> Plank's constant is 662.61×10^{36}J-s. Reduced plank's constant is 105.46×10^{36}J-s. Boltzmann's constant is 13.806×10^{24}J/K. Elementary charge is 160.22×10^{21}C. Speed of light is 299.79Mm/s. Zero degrees celsius is 273.15K. Permittivity of free space is 8.8542pF/m. Permeability of free space is 1.2566µH/m. Absolute zero is 273.15°C.

7.3.6 Timeit Example

A Python module that benefits from *QuantiPhy* is *timeit*, a package in the standard library that runs a code snippet a number of times and prints the elapsed time for the test. However, from a usability perspective it has several issues. First, it prints out the elapsed time of all the repetitions rather than dividing the elapsed time by the number of repetitions and reporting the average time per operation. So it can quickly allow you to compare the relative speed of various operations, but it does not directly give you a sense of the time required in absolute terms. Second, it does not label its output, so it is not clear what is being displayed. Here is an example where *timeit* has been fortified with *QuantiPhy* to make the output more readable. To make it more interesting, the timing results are run on *QuantiPhy* itself. The results give you a feel for how much slower *QuantiPhy* is to both convert strings to quantities and quantities to strings compared into the built-in float class.

```
#!/usr/bin/env python3
from timeit import timeit
from random import random, randint
from quantiphy import Quantity

# preferences
trials = 100_000
Quantity.set_prefs(
    prec = 2,
    show_label = True,
    label_fmt = '{n:>40}: {v}',
    map_sf = Quantity.map_sf_to_greek
)

# build the raw data, arrays of random numbers
s_numbers = []
s_quantities = []
numbers = []
```

(continues on next page)

(continued from previous page)

```
quantities = []
for i in range(trials):
    mantissa = 20*random()-10
    exponent = randint(-35, 35)
    number = '%0.25fe%s' % (mantissa, exponent)
    quantity = number + ' Hz'
    s_numbers.append(number)
    s_quantities.append(quantity)
    numbers.append(float(number))
    quantities.append(Quantity(number, 'Hz'))

# define testcases
testcases = [
    '[float(v) for v in s_numbers]',
    '[Quantity(v) for v in s_quantities]',
    '[str(v) for v in numbers]',
    '[str(v) for v in quantities]',
]

# run testcases and print results
print(f'For {Quantity(trials)} values ...')
for case in testcases:
    elapsed = timeit(case, number=1, globals=globals())
    result = Quantity(elapsed/trials, units='s/op', name=case)
    print(result)
```

The results are:

```
For 100k iterations ...
           [float(v) for v in s_numbers]: 638 ns/op
     [Quantity(v) for v in s_quantities]: 15.3 µs/op
              [str(v) for v in numbers]: 1.03 µs/op
            [str(v) for v in quantities]: 28.1 µs/op
```

You can see that *QuantiPhy* is considerably slower than the float class, which you should be aware of if you are processing large quantities of numbers.

Contrast this with the normal output from *timeit*:

```
0.05213119700783864
1.574107409993303
0.10471829099697061
2.3749650190002285
```

The essential information is there, but it takes longer to make sense of it.

7.3.7 Disk Usage Example

Here is a simple example that uses *QuantiPhy* to clean up the output from the Linux disk usage utility. It runs the *du* command, which prints out the disk usage of files and directories. The results from *du* are gathered and then sorted by size and then the size and name of each item is printed.

Quantity is used to scale the filesize reported by *du* from KB to B. Then the list of files is sorted by size. Here we are exploiting the fact that quantities act like floats, and so the sorting can be done with no extra effort. Finally, the ability to render to a number with a scale factor and units is used when presenting the results.

```
#!/usr/bin/env python3
# runs du and sorts the output while suppressing any error messages from du

from quantiphy import Quantity
from inform import display, fatal, os_error
from shlib import Run
import sys

try:
    du = Run(['du', '-xd1'] + sys.argv[1:], modes='sWEO1')

    files = []
    for line in du.stdout.splitlines():
        if line:
            size, _, filename = line.partition('\t')
            files += [(Quantity(size, scale=(1024, 'B')), filename)]

    files.sort(key=lambda x: x[0])

    for size, name in files:
        display('{:8.2b}  {}'.format(size, name))

except OSError as err:
    fatal(os_error(err))
except KeyboardInterrupt:
    display('dus: killed by user.')
```

And here is an example of the programs output:

```
 460 KiB  quantiphy/examples/delta-sigma
 464 KiB  quantiphy/examples
1.54 KiB  quantiphy/doc
3.48 MiB  quantiphy
```

7.3.8 Parameterized Simulation Example

In this example, Python is used to perform a simulation of a modulator. There are a collection of parameters that control the simulation, which are placed at the top of the Python file as documentation. *Quantity.extract()* is used to access these parameters and control the simulation. In this way, modifying the simulation parameters is easy and the documentation is always up to date.

```
#!/usr/bin/env python3

r"""
Simulates a second-order  modulator with the following parameter values:

    Fclk = 50MHz            -- clock frequency
    Fin = 200kHz            -- input frequency
    Vin = 950mV             -- input voltage amplitude (peak)
    gain1 = 0.5             -- gain of first integrator
    gain2 = 0.5             -- gain of second integrator
    Vmax = 1V               -- quantizer maximum input voltage
    Vmin = -1V              -- quantizer minimum input voltage
    # levels = 16             -- quantizer output levels
    levels = 4              -- quantizer output levels
    Tstop = 1/Fin "s"       -- simulation stop time
    Tstart = -0.5/Fin "s"   -- simulation start time (points with t<0 are discarded)
    vin_file = 'vin.wave'   -- output data file for vin
    vout_file = 'vout.wave' -- output data file for vout
    dout_file = 'dout.wave' -- output data file for dout
"""
# The values given above are used in the simulation, no further modification
# of the code given below is required when changing these parameters.

from quantiphy import Quantity
from math import sin, tau
from inform import display, error, os_error

class Integrator:
    def __init__(self, gain=1):
        self.state = 0
        self.gain = gain

    def update(self, vin):
        self.state += self.gain*vin
        return self.state


class Quantizer:
    def __init__(self, v_max, v_min, levels):
        self.v_min = v_min
        self.levels = levels
        self.delta = (v_max - v_min)/(levels - 1)

    def update(self, v_in):
        level = (v_in - self.v_min) // self.delta
        level = 0 if level < 0 else level
```

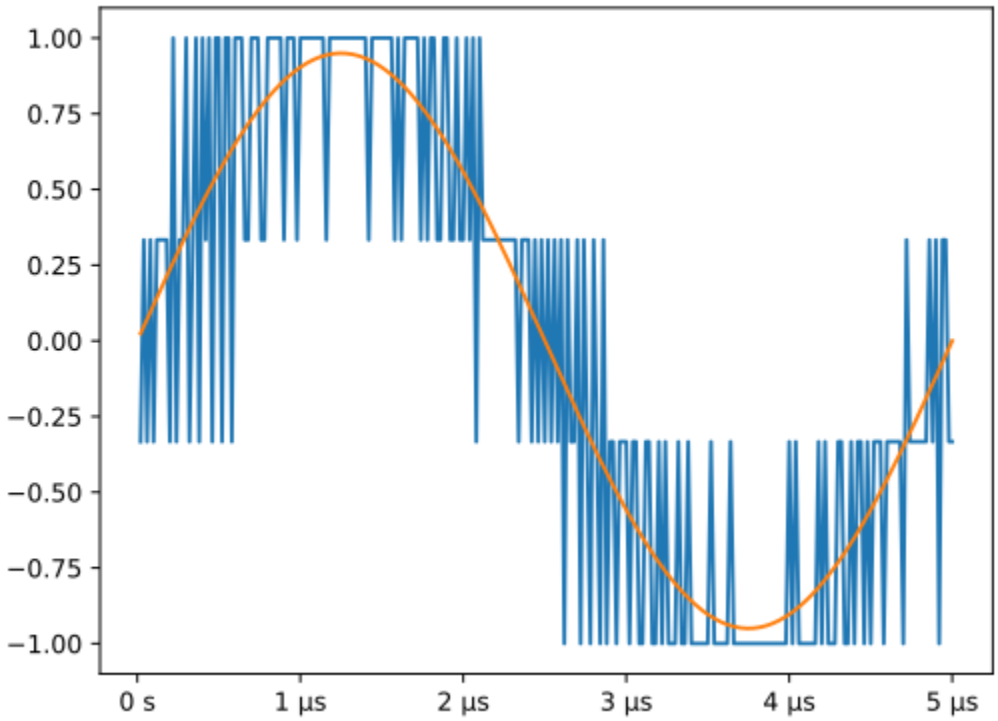
(continues on next page)

(continued from previous page)

```
        level = self.levels-1 if level >= self.levels else level
        return int(level), self.delta*level + self.v_min


class Source:
    def __init__(self, f_in, amp):
        self.omega = tau*f_in
        self.amp = amp

    def update(self, t):
        return self.amp*sin(self.omega*t)


# read simulation parameters and load into module namespace
parameters = Quantity.extract(__doc__)
globals().update(parameters)

# display the simulation parameters
display('Simulation parameters:')
for k, v in parameters.items():
    try:
        display(f'   ', v.render(show_label='f'))
    except AttributeError:
        display(f'    {k} = {v}')

# instantiate components
integrator1 = Integrator(gain1)
integrator2 = Integrator(gain2)
quantizer = Quantizer(Vmax, Vmin, levels)
sine = Source(Fin, Vin)

# run simulation
t = Tstart
dt = 1/Fclk
v_out = 0
t_stop = Tstop
try:
    fvin = open(vin_file, 'w')
    fvout = open(vout_file, 'w')
    fdout = open(dout_file, 'w')
    while t < t_stop:
        v_in = sine.update(t)
        v_int1 = integrator1.update(v_in - v_out)
        v_int2 = integrator2.update(v_int1 - v_out)
        d_out, v_out = quantizer.update(v_int2)
        if (t >= 0):
            print(t, v_in, file=fvin)
            print(t, v_out, file=fvout)
            print(t, d_out, file=fdout)
        t += dt
except OSError as e:
    error(os_error(e))
```

Notice that *levels* was specified twice, but the first proceeded by # causing it to be ignored.

The output of this example can be used as the input to the next. With these parameters, it produces this waveform:



7.3.9 MatPlotLib Example

In this example *QuantiPhy* is used to create easy to read axis labels in MatPlotLib. It uses NumPy to do a spectral analysis of a signal and then produces an SVG version of the results using MatPlotLib.

```
#!/usr/bin/env python3

import numpy as np
from numpy.fft import fft, fftfreq, fftshift
import matplotlib as mpl
mpl.use('SVG')
from matplotlib.ticker import FuncFormatter
import matplotlib.pyplot as pl
from quantiphy import Quantity
Quantity.set_prefs(map_sf=Quantity.map_sf_to_sci_notation)

# read the data from delta-sigma.smpl
data = np.fromfile('delta-sigma.smpl', sep=' ')
time, wave = data.reshape((2, len(data)//2), order='F')

# print out basic information about the data
```

(continues on next page)

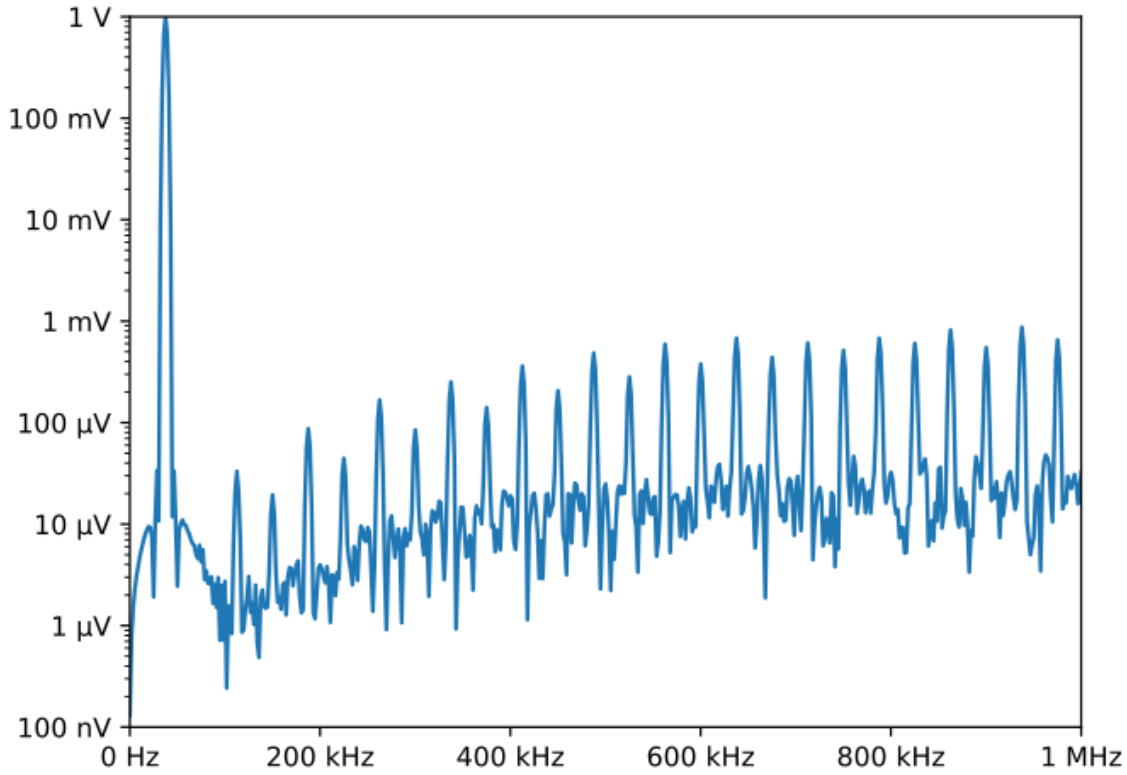
(continued from previous page)

```
timestep = Quantity(time[1] - time[0], name='Time step', units='s')
nonperiodicity = Quantity(wave[-1] - wave[0], name='Nonperiodicity', units='V')
points = Quantity(len(time), name='Time points')
period = Quantity(timestep * len(time), name='Period', units='s')
freq_res = Quantity(1/period, name='Frequency resolution', units='Hz')
with Quantity.prefs(show_label=True, prec=2):
    print(timestep, nonperiodicity, points, period, freq_res, sep='\n')

# create the window
window = np.kaiser(len(time), 11)/0.37
    # beta=11 corresponds to alpha=3.5 (beta = pi*alpha)
    # the processing gain with alpha=3.5 is 0.37
windowed = window*wave

# transform the data into the frequency domain
spectrum = 2*fftshift(fft(windowed))/len(time)
freq = fftshift(fftfreq(len(wave), timestep))

# define the axis formatting routines
freq_formatter = FuncFormatter(lambda v, p: str(Quantity(v, 'Hz')))
volt_formatter = FuncFormatter(lambda v, p: str(Quantity(v, 'V')))

# generate graphs of the resulting spectrum
fig = pl.figure()
ax = fig.add_subplot(111)
ax.plot(freq, np.absolute(spectrum))
ax.set_yscale('log')
ax.xaxis.set_major_formatter(freq_formatter)
ax.yaxis.set_major_formatter(volt_formatter)
pl.savefig('spectrum.svg')
ax.set_xlim((0, 1e6))
ax.set_ylim((1e-7, 1))
pl.savefig('spectrum-zoomed.svg')
```

This script produces the following textual output:

```
Time step = 20 ns
Nonperiodicity = 2.3 pV
Time points = 28k
Period = 560 µs
Frequency resolution = 1.79 kHz
```

And the following is one of the two graphs produced:



Notice the axis labels in the generated graph. Use of *QuantiPhy* makes the widely scaled units compact and easy to read.

MatPlotLib provides the EngFormatter that you can use as an alternative to *QuantiPhy* for formatting your axes with SI scale factors, which also provides the *format_eng* function for converting floats to strings formatted with SI scale factors and units. So if your needs are limited, as they are in this example, that is generally a good way to go. One aspect of *QuantiPhy* that you might prefer is the way it handles very large or very small numbers. As the numbers get either very large or very small *EngFormatter* starts by using unfamiliar scale factors (*YZPEzy*) and then reverts to e-notation. *QuantiPhy* allows you to control whether to use unfamiliar scale factors but does not use them by default. It also can be configured to revert to engineering scientific notation (ex: $13.806{\times}10^{24}$ J/K) when no scale factors are appropriate. Though not necessary for this example, that was done above with the line:

```
Quantity.set_prefs(map_sf=Quantity.map_sf_to_sci_notation)
```

7.3.10 Flicker Noise

This example represents a very typical use of *QuantiPhy* in a simulation script. As in the two previous examples, it includes both extraction of simulation parameters from the script's documentation and attractive formatting of units in MatPlotLib graphs. It is a bit long and you cannot run it yourself as it requires access to a proprietary circuit simulator, and as such the code is not included here. But it is an excellent example of how to use *QuantiPhy* in a variety of ways. You can find the Flicker Noise code on GitHub. It produces results like the following:

7.3.11 Cryptocurrency Example

This example displays the current price of various cryptocurrencies and the total value of a hypothetical portfolio of currencies. *QuantiPhy* performs conversions from the prices of various currencies to dollars. The latest prices are downloaded from cryptocompare.com. A summary of the prices is printed and then they are multiplied by the portfolio holdings to find the total worth of the portfolio, which is also printed.

It demonstrates some of the features of *UnitConversion*.

```
#!/usr/bin/env python3

import requests
from inform import display, fatal, os_error, terminate
from quantiphy import Quantity, UnitConversion, InvalidNumber
Quantity.set_prefs(prec=2)

# read holdings
try:
    with open('holdings') as f:
        lines = f.read().splitlines()
    holdings = {
        q.units: q for q in [
            Quantity(l, ignore_sf=True) for l in lines if l
        ]
    }
except OSError as e:
    fatal(os_error(e))
except InvalidNumber as e:
    fatal(e)

# download latest asset prices from cryptocompare.com
currencies = dict(
    fsyms = ','.join(holdings.keys()),  # from symbols
    tsyms = 'USD',                      # to symbols
)
url_args = '&'.join(f'{k}={v}' for k, v in currencies.items())
base_url = f'https://min-api.cryptocompare.com/data/pricemulti'
url = '?'.join([base_url, url_args])
try:
    response = requests.get(url)
except KeyboardInterrupt:
    terminate('Killed by user.)
except Exception as e:
    fatal('cannot connect to cryptocompare.com.')
conversions = response.json()

# define unit conversions
converters = {
    sym: UnitConversion(('$', 'USD'), sym, conversions[sym]['USD'])
    for sym in holdings
}

# sum total holdings
total = Quantity(sum(q.scale('$') for q in holdings.values()), '$')
```

(continues on next page)

(continued from previous page)

```
# show summary of holdings and conversions
for sym, q in holdings.items():
    value = f'{q:>9q} = {q:<7q$} {100*q.scale("$")/total:.0f}%'
    price = f'1 {sym} = {converters[sym].convert()}'
    display(f'{value:<25s} ({price})')
display(f'    Total = {total:q}')
```

This script reads a file 'holdings' that contains the number of tokens you hold of each of your cryptocurrencies. That file would contain one currency per line and look like this:

```
10 BTC
100 ETH
100 BCH
100 ZEC
10,000 EOS
100,000 ADA
```

The output of the script looks like this:

```
  10 BTC = $65.8k  30%   (1 BTC = $6.58k)
 100 ETH = $22.4k  10%   (1 ETH = $224)
 100 BCH = $51.5k  24%   (1 BCH = $515)
 100 ZEC = $12.7k  6%    (1 ZEC = $127)
 10 kEOS = $57.6k  26%   (1 EOS = $5.76)
100 kADA = $8.16k  4%    (1 ADA = $81.6m)
   Total = $218k
```

If you prefer the output in fixed-point format, you can replace the last part of this code with:

```
# show summary of holdings and conversions
for sym, q in holdings.items():
    value = f'{q:>10.2p} = {q:>#11,.2p$}  {100*q.scale("$")/total:,.0f}%'
    price = f'1 {sym} = {converters[sym].convert():>#9,.2p}'
    display(f'{value:<30s} ({price})')
display(f'     Total = {total:>#11,.2p}')
```

If you do, the output of the script looks like this:

```
    10 BTC =  $65,847.10  30%  (1 BTC = $6,584.71)
   100 ETH =  $22,401.00  10%  (1 ETH =   $224.01)
   100 BCH =  $51,450.00  24%  (1 BCH =   $514.50)
   100 ZEC =  $12,726.00  6%   (1 ZEC =   $127.26)
 10000 EOS =  $57,600.00  26%  (1 EOS =     $5.76)
100000 ADA =   $8,203.00  4%   (1 ADA =     $0.08)
     Total = $218,227.10
```

A more sophisticated version of cryptocurrency this example can be found on GitHub.

7.3.12 Dynamic Unit Conversions

Normally unit conversions are static, meaning that once the conversion values are set they do not change during the life of the process. However, that need not be true if functions are used to perform the conversion. In the following example, the current price of Bitcoin is queried from a price service and used in the conversion. The price service is queried each time a conversion is performed, so it is always up-to-date, no matter how long the program runs.

```
#!/usr/bin/env python3

# Bitcoin
# This example demonstrates how to use UnitConversion to convert between
# bitcoin and dollars at the current price.

from quantiphy import Quantity, UnitConversion
import requests

# get the current bitcoin price from coingecko.com
url = 'https://api.coingecko.com/api/v3/simple/price'
params = dict(ids='bitcoin', vs_currencies='usd')
def get_btc_price():
    try:
        resp = requests.get(url=url, params=params)
        prices = resp.json()
        return prices['bitcoin']['usd']
    except Exception as e:
        print('error: cannot connect to coingecko.com.')

# use UnitConversion from QuantiPhy to perform the conversion
bitcoin_units = ['BTC', 'btc', '', '']
satoshi_units = ['sat', 'sats', 's']
dollar_units = ['USD', 'usd', '$']
UnitConversion(
    dollar_units, bitcoin_units,
    lambda b: b*get_btc_price(), lambda d: d/get_btc_price()
)
UnitConversion(satoshi_units, bitcoin_units, 1e8)
UnitConversion(
    dollar_units, satoshi_units,
    lambda s: s*get_btc_price()/1e8, lambda d: d/(get_btc_price()/1e8),
)

unit_btc = Quantity('1 BTC')
unit_dollar = Quantity('$1')

print(f'{unit_btc:>8,.2p} = {unit_btc:,.2p$}')
print(f'{unit_dollar:>8,.2p} = {unit_dollar:,.0psat}')
```

When run, the script prints something like this:

```
1 BTC = $17,211.91
   $1 = 5,810 sat
```

7.4 Accessories

A collection utility programs have been developed that employ *QuantiPhy* to enhance their functionality. These utilities are not included as part of *QuantiPhy*, but are available via PyPi.

7.4.1 Engineering Calculator

ec is a handy command-line calculator for engineers and scientists that employs Reverse-Polish Notation (RPN) and allows numbers to be specified with units and SI scale factors. With RPN, the arguments are pushed onto a stack and the operators pull the needed argument from the stack and push the result back onto the stack. For example, to compute the effective resistance of two parallel resistors:

```
> ec
0: 100k 50k ||
33.333k:
```

And here is a fuller example that shows some of the features of *ec*. In this case we create initialization scripts, ~/.ecrc and ./.ecrc, and a dedicated script, compute-zo, and use it to compute the output impedance of a simple RC circuit:

```
> cat ~/.ecrc
# define some functions useful in phasor analysis
(2pi * "rads/s")to_omega    # convert frequency in Hertz to radians/s
(mag 2pi / "Hz")to_freq     # convert frequency in radians/s to Hertz
(j2pi * "rads/s")to_jomega  # convert frequency in Hertz to imaginary radians/s

> cat ./.ecrc
# define default values for parameters
10MHz =freq   # operating frequency
1nF =Cin      # input capacitance
50 =Rl       # load resistance

> cat ./compute-zo
freq to_jomega           # enter 10MHz and convert to radial freq.
Cin * recip              # enter 1nF, multiply by  and reciprocate
                         # to compute impedance of capacitor at 10MHz
Rl ||                    # enter 50 Ohms and compute impedance of
                         # parallel combination
"" =Zo                  # apply units of  and save to Zo
ph                       # compute the phase of impedance
Zo mag                   # recall complex impedance from Zo and compute its magnitude
`Zo = $0  $1 @ $freq.`  # display the magnitude and phase of Zo
quit

> ec compute-zo
Zo = 15.166   -72.343 degs @ 10 MHz.

> ec 500pF =Cin compute-zo
Zo = 26.851   -57.518 degs @ 10 MHz.
```

It may be a bit confusing, just remember that with RPN you give the values first by pushing them on to the stack, and then act on them. And once you get use to it, you'll likely find it quite efficient.

The source code is available from the ec repository on GitHub, or you can install it directly with:

```
pip install --user engineering_calculator
```

7.4.2 Time-Value of Money

Time-Value of Money (*TVM*) is a command line program that is used to perform calculations involving interest rates. It benefits from *QuantiPhy* in that it allows values to be given quite flexibly and concisely. The goal of the program is to allow you to quickly run what-if experiments involving financial calculations. So the fact that *QuantiPhy* allows the user to type 1.2M rather than 1200000 or 1.2e6 helps considerably to reach that goal. For example, when running the program, this is what you would type to calculate the monthly payments for a mortgage:

```
tvm  -p -250k -r 4.5 pmt
```

The program would respond with:

```
pmt = $1,266.71
pv = -$250,000.00
fv = $0.00
r = 4.5%
N = 360
```

The act of converting strings to numbers on the way in and converting numbers to strings on the way out is performed by *QuantiPhy*.

QuantiPhy is quite flexible when it comes to converting a string to a number, so the present value can be given in any of the following ways: -$250k, -$250,000, -$2.5e5. You can also specify the value without the currency symbol, which is desirable as it generally confuses the shell.

The source code is available from the tvm repository on GitHub, or you can install it directly with:

```
pip install --user tvm
```

7.4.3 PSF Utils

PSF Utils is a library that allows you to read data from a Spectre PSF ASCII file. Spectre is a commercial circuit simulator produced by Cadence Design Systems. PSF files contain signals generated by Spectre. This package also contains two programs that are useful in their own right, but also act as demonstrators as to how to use the library. They are *list-psf* and *plot-psf*. The first lists the available signals in a file, and the other displays them.

QuantiPhy is used by *plot-psf* when generating the axis labels.

The source code is available from the psf_utils repository on GitHub, or you can install it directly with:

```
pip install --user psf_utils
```

7.4.4 Evaluate Expressions in Strings

QuantiPhy Eval is yet another calculator, this one is a Python API that allows you to evaluate expressions that contain numbers with units and SI scale factors that are embedded in strings.

```
>>> from quantiphy_eval import evaluate

>>> avg_price = evaluate('($1.2M + $1.3M)/2', '$')
>>> print(avg_price)
$1.25M
```

The source code is available from the quantiphy_eval repository on GitHub, or you can install it directly with:

```
pip install --user quantiphy_eval
```

7.4.5 Schedule Reminders

remind is command line reminder program. At the appointed time it sends you a notification to remind you of some of event. Such a program has no need for SI scale factors. Instead, this program uses the ability of *QuantiPhy* to scale numbers based on their units to provide a user-interface that takes convenient descriptions of time intervals such as 20m or 2h.

```
> remind 45m remove roast from oven
Alarm scheduled for 6:36 PM, 45 minutes from now.
Message: remove roast from oven
```

You can specify the time as either a time-of-day or an elapsed time. You can even combine them to do simple calculations:

```
> remind 10am -15m meet with Jamie
Alarm scheduled for 9:45 AM, 108 minutes from now.
Message: meet with Jamie
```

The source code is available from the remind repository on GitHub, or you can install it directly with:

```
pip install --user schedule-reminder
```

7.4.6 RKM Codes

RKM codes are a way of writing numbers that is often used for specifying the sizes of resistors and capacitors on schematics and on the components themselves. In RKM codes the radix is replaced by the scale factor and the units are suppressed. Doing so results in a compact representation that is less likely to be misinterpreted if the number is poorly rendered. For example, a 6.8K could be read as 68K if the decimal point is somehow lost. The RKM version of 6.8K is 6K8. RKM codes are described on Wikipedia.

The popularity of RKM codes was fading because they address a problem that is less common today. However they are making something of a come back as all the characters in a RKM code are either letters or digits and so they can be embedded in a software identifier without introducing illegal characters.

```
>>> from rkm_codes import from_rkm, to_rkm

>>> r = from_rkm('6K8')
```

(continues on next page)

(continued from previous page)

```
>>> r
Quantity('6.8k')

>>> to_rkm(r)
'6K8'
```

As a practical example of the use of RKM codes, imagine wanting a program that creates pin names for an electrical circuit based on a naming convention where the pin names must be valid identifiers (must consist only of letters, digits, and underscores). It would take a table of pin characteristics that are used to create the names.

For example:

```
>>> from quantiphy import Quantity
>>> from rkm_codes import to_rkm, set_prefs as set_rkm_prefs

>>> pins = [
...     dict(kind='ibias', direction='out', polarity='sink', dest='dac', value='250nA'),
...     dict(kind='ibias', direction='out', polarity='src', dest='rampgen', value='2.5µA
↪'),
...     dict(kind='vref', direction='out', dest='dac', value='1.25V'),
...     dict(kind='vdda', direction='in', value='2.5V'),
... ]
>>> set_rkm_prefs(map_sf={}, units_to_rkm_base_code=None)

>>> for pin in pins:
...     components = []
...     if 'value' in pin:
...         pin['VALUE'] = to_rkm(Quantity(pin['value']))
...     for name in ['dest', 'kind', 'direction', 'VALUE', 'polarity']:
...         if name in pin:
...             components.append(pin[name])
...     print('_'.join(components))
dac_ibias_out_250n_sink
rampgen_ibias_out_2u5_src
dac_vref_out_1v2
vdda_in_2v5
```

The source code is available from the rkm_codes repository on GitHub, or you can install it directly with:

```
pip install --user rkm_codes
```

7.5 Releases

7.5.1 Latest development release

Version: 2.20
Released: 2024-04-27

7.5.2 2.20 (2024-04-27)

- Include full quantities if available in *IncompatibleUnits* errors

7.5.3 2.19 (2023-01-05)

- Added new standard SI scale factors (Q, R, r, q).
- Subclasses of *Quantity* with units now convert values to the desired units rather than allowing the units of the class to be overridden by those of the value.
- Added scale factor conversion.
- Added quantity functions: *as_real()*, *as_tuple()*, *render()*, *fixed()*, and *binary()*.
- Fixed rendering of full precision numbers in *Quantity.fixed()*.
- Added *preferred_units* *Quantity* preference.
- Added "cover" option to *strip_radix* *Quantity* preference.
- Added type hints.

7.5.4 2.18 (2022-08-31)

- Support parametrized unit conversions (such as molarity).
- Allow % to act as a scale factor.
- First argument of scaling functions are now guaranteed to be quantities.
- Added *UnitConversion.fixture()* decorator function.
- Added *UnitConversion.activate()* method (allows overridden converters to be re-activated).

7.5.5 2.17 (2022-04-04)

- Refine the list of currency symbols.
- Allows currency symbols to be given before or after the underlying number.
- Allow *Quantity* subclasses to be used in scaling if they have units.

7.5.6 2.16 (2021-12-14)

- Add support for — as comment character and make it the default.

7.5.7 2.15 (2021-08-03)

- Updated predefined physical constants to CODATA 2018 values.
- Switched to more permissive MIT license.
- Add feet to the available length/distance unit conversions.

7.5.8 2.14 (2021-06-18)

- Allow primary argument of *Quantity.is_close()* and *Quantity.add()* to be a string.

7.5.9 2.13 (2020-10-13)

- Allow currency symbols in compound units (ex: $/oz or lbs/$).

7.5.10 2.12 (2020-07-25)

- Bug fix release.

7.5.11 2.11 (2020-07-19)

- Dropping support for all versions of Python older than 3.5.
- Added *sia* form (ASCII only SI scale factors).
- Added *only_e_notation* argument to *Quantity.all_from_conv_fmt()*.
- Added *Quantity.reset_prefs()* method.

7.5.12 2.10 (2020-03-2)

- Added *negligible*, *tight_units*, *nan*, and *inf* preferences.
- Added *negligible* argument to render.
- Added *infinity_symbol* attribute.
- Changed the return values for *Quantity.is_nan()* and *Quantity.is_infinite()*.

7.5.13 2.9 (2020-01-28)

- Made *Quantity.extract()* more forgiving.
- Support radix and comma processing when converting strings to *Quantity*.

7.5.14 2.8 (2020-01-08)

- Fix nit in installer (setup.py).

7.5.15 2.7 (2019-12-17)

- Improve the ability of both *Quantity.add()* and *Quantity.scale()* to retain attributes.
- Added *accept_binary* preference.
- Support all preferences as class attributes.
- Allow radix and comma to be replaced by adding *radix* and *comma* preferences.

7.5.16 2.6 (2019-09-24)

- Now support Quantity arguments with *Quantity.extract()*.
- Allow plus and minus signs to be replaced with Unicode equivalents.

7.5.17 2.5 (2019-01-16)

- Added RKM codes example.
- Added *check_value* = 'strict' to *Quantity.add()*.
- Added backward compatibility for *form* argument of *Quantity.render()* if it is passed as unnamed argument.
- Made *Quantity.extract()* a bit more general.
- Reformulated exceptions.
- Added support for binary scale factors and *Quantity.binary()*.

7.5.18 2.4 (2018-09-12)

- Fixed bug in format that resulted in several format codes ignoring width
- Follow Python convention of right-justifying numbers by default.
- Add Quantity.add() (adds a number to a quantity returning a new quantity)
- Added # alternate form of string formatting.
- Change *show_si* to *form* (argument on *Quantity.set_prefs()* and *Quantity.render()* (*show_si* is now obsolete, use *form='si'* instead).
- Added concept of equivalent units for unit conversion to documentation.
- Enhance UnitConversion so that it supports nonlinear conversions.

7.5.19 2.3 (2018-03-11)

- Enhanced *Quantity.extract()*
  - non-conforming lines are now ignored
  - values may be expressions
  - values need not be quantities
  - can specify a quantity name distinct from dictionary name
- Enhanced the formatting capabilities.
  - added center alignment
  - added *p* format
  - added *show_commas* preference.
  - added *strip_zeros*, *strip_radix* to *Quantity.render()*
  - added *Quantity.fixed()* method
  - added *Quantity.format()* method
  - support any format specifier supported by Python for floats

7.5.20 2.2 (2017-11-22)

- Added *Quantity.scale()*
- Added *UnitConversion.convert()*
- Added *strip_zeros*
- Added no-op conversions (units change but value stays the same, ex: $ → USD)

7.5.21 2.1 (2017-07-30)

The primary focus of this release was on improving the documentation, though there are a few small feature enhancements.

- Added support for SI standard composite units
- Added support for non-breaking space as spacer
- Removed constraint in *Quantity.extract()* that names must be identifiers

7.5.22 2.0 (2017-07-15)

This is a 'coming of age' release where the emphasis shifts from finding the right interface to providing an interface that is stable over time. This release includes the first formal documentation and a number of new features and refinements to the API.

- Created formal documentation
- Enhanced *label_fmt* to accept {V}
- Allow quantity to be passed as value to *Quantity*
- Replaced *Quantity.add_to_namespace* with *Quantity.extract()*

- Raise *NameError* rather than *AssertionError* for unknown preferences
- Added `Quantity.all_from_conv_fmt()` and `Quantity.all_from_si_fmt()`
- Change *assign_rec* to support more formats
- Changed *Constant()* to `add_constant()`
- Changed the way preferences are implemented
- Changed name of preference methods: *set_preferences* → *set_prefs*, *get_preference* → *get_pref*
- Added `Quantity.prefs()` (preferences context manager)
- Split *label_fmt* preference into two: *label_fmt* and *label_fmt_full*
- Added *show_desc* preference
- Allow *show_label* to be either 'a' or 'f' as well True or False
- Renamed *strip_dp* option to *strip_radix*
- Added *number_fmt* option

7.5.23 1.3 (2017-03-19)

- Reworked constants
- Added unit systems for physical constants

7.5.24 1.2 (2017-02-24)

- Allow digits after decimal point to be optional
- Support underscores in numbers
- Allow options to be monkey-patched on to Quantity objects
- Add *strip_dp* option
- Fix some issues in full precision mode
- Ranamed some options, arguments and methods

7.5.25 1.1 (2016-11-27)

- Added *known_units* preference.
- Added *get_preference* class method.

7.5.26 1.0 (2016-11-26)

- Initial production release.
- genindex

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

P

Q

R

S

T

U

Z